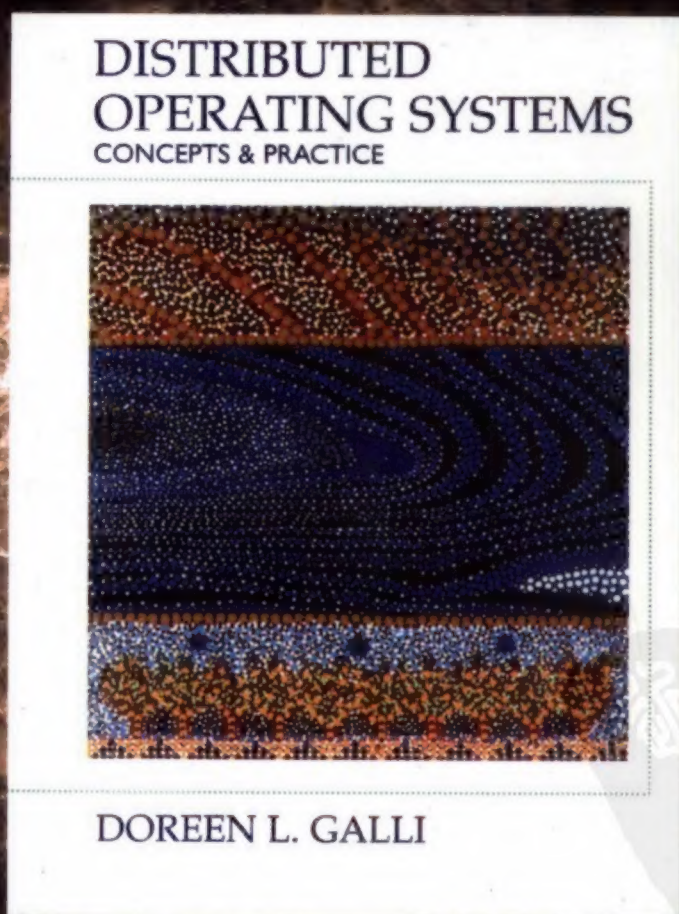


计 算 机 科 学 丛 书

分布式操作系统

原理与实践

(美) Doreen L. Galli 著 徐良贤 唐英 毛家菊 金恩华 等译



Distributed Operating Systems
Concepts and Practice



机械工业出版社
China Machine Press

Prentice
Hall

本书从概念和实践的角度详细论述了分布式操作系统的各个主要方面，包含广泛的算法的具体实例，提供可以实践的面向项目的练习，本书提及的核心Web站点、ftp站点和文献提供了大量参考资料，并且包含了实际操作系统的相关范例，以加深概念和展示分布式系统设计人员所需进行的设计与操作。最后，灵活的设计使本书可用于教学和技术培训。

本书的教学特点如下：

- 供进一步深入理解的详细说明，包含诸如复杂算法和更深入的例子等内容。
- Windows 2000案例研究，展示一个实际的商业解决方案。
- 面向项目的习题提供有关经验。
- 参考资料包括：
 - A. 供进一步学习的综述资料
 - B. 研究论文
 - C. 核心Web站点和ftp站点
- 一个简化的分布式应用程序，以展示分布式程序设计的关键概念。
- 缩写词的完整列表，以帮助阅读并为快速查询提供一个集中存放地点。
- 本书的Web站点为www.prenhall.com/galli

作者简介

Doreen L. Galli

博士现在在亚特兰大从事远程通信的研究工作，是分布式计算和系统集成的专家。同时，作为计算机科学的教授，她曾讲授过关于高级系统和网络技术的许多课程。她在Waterloo大学获得了博士学位。

ISBN 7-111-10952-X



华章图书

网上购书: www.china-pub.com

北京市西城区百万庄南街1号 100037
购书热线: (010)68995259, 8006100280 (北京地区)
总编信箱: chiefeditor@hzbook.com

ISBN 7-111-10952-X/TP · 2614
定价: 38.00 元



计 算 机 科 学 丛 书

TP316.4

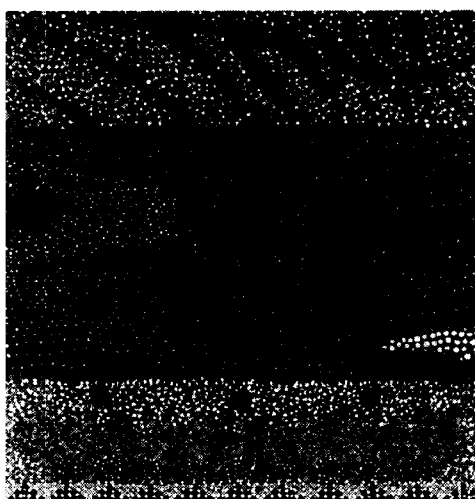
J28

分布式操作系统

原理与实践

(美) Doreen L. Galli 著 徐良贤 唐英 毛家菊 金恩华 等译

DISTRIBUTED
OPERATING SYSTEMS
CONCEPTS & PRACTICE



DOREEN L. GALLI



A1016105

Distributed Operating Systems
Concepts and Practice



机械工业出版社
China Machine Press

本书从概念和实践的角度详细论述了分布式操作系统的各个主要方面,还包含了实际操作系统的相关范例以及广泛算法的具体实例,其中提及的核心 Web 站点、ftp 站点和文献提供了大量参考资料。此外,Windows 2000 案例研究提供了一个实际商业解决方案的例子,附录中还有一个简单的 C/S 实际应用来演示关键的分布式计算程序设计概念,以加深概念并展示分布式操作系统设计人员所需进行的设计与操作。

本书内容丰富,结构合理,适于作为计算机及相关专业的本科生和研究生的教材,也是计算机从业人员掌握分布式操作系统原理的理想读物。

Doreen L. Galli: Distributed Operating Systems: Concepts & Practice (ISBN 0-13-079843-6).

Authorized translation from the English language edition published by Prentice Hall PTR.

Copyright © 2000 by Prentice Hall PTR, Inc.

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2002 by China Machine Press.

本书中文简体字版由美国 Prentice Hall PTR 公司授权机械工业出版社独家出版。未经出版者书面许可,不得以任何方式复制或抄袭本书内容。

版权所有,侵权必究。

本书版权登记号: 图字: 01-2001-2205

图书在版编目 (CIP) 数据

分布式操作系统: 原理与实践/ (美) 加利 (Galli, D.L.) 著; 徐良贤等译. - 北京: 机械工业出版社, 2003.1

(计算机科学丛书)

书名原文: Distributed Operating Systems: Concepts and Practice

ISBN 7-111-10952-X

I. 分… II. ①加… ②徐… III. 分布式操作系统 - 原理 IV. TP316

中国版本图书馆 CIP 数据核字 (2002) 第 070382 号

机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑: 杨文

北京第二外国语学院印刷厂印刷·新华书店北京发行所发行

2003 年 1 月第 1 版第 1 次印刷

787mm×1092mm 1/16·21.75 印张

印数: 0 001~5 000 册

定价: 38.00 元

凡购本书, 如有倒页、脱页、缺页, 由本社发行部调换

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域中取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall, Addison-Wesley, McGraw-Hill, Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum, Stroustrup, Kernighan, Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及度藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：针对本科生的核心课程，剔抉外版菁华而成“国外经典教材”系列；对影印版的教材，则单独开辟出“经典原版书库”；定位在高级教程和专业参考的“计算机科学丛书”还将保持原来的风格，继续出版新的品种。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

“国外经典教材”是响应教育部提出的使用外版教材的号召，为国内高校的计算机本科教学度身订造的。在广泛地征求并听取丛书的“专家指导委员会”的意见后，我们最终选定了这20多种篇幅内容适度、讲解鞭辟入里的教材，其中的大部分已经被M.I.T.、Stanford、U.C. Berkley、C.M.U.等世界名牌大学采用。丛书不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzedu@hzbook.com

联系电话：(010) 68995265

联系地址：北京市西城区百万庄南街1号

邮政编码：100037

译者序

近年来，分布式操作系统在国内外都获得了迅猛的发展，成为计算机科学与技术研究领域中的一个热点，同时在商业应用方面也取得了丰硕的成果。学习和研究世界上分布式系统最新的理论和技术，以便在该领域迅速跟上并超越国际水平，是我国计算机界义不容辞的任务。因此我们翻译了 Doreen L. Galli 博士的《分布式操作系统：原理与实践》一书。

本书涵盖了分布式操作系统的所有内容，全面地介绍了分布式系统环境中的内核、通信、内存管理、并发控制等基本概念和算法，并对进程管理、文件系统、事务管理和同步等方面的高级概念和算法进行了详细的介绍和研究。

本书紧密结合当前的最新技术，并对这些技术进行了详细的阐述，比如基于对象的分布式操作系统、分布式操作系统的安全等，同时还给出了具有实际意义的研究实例——Windows 2000。它不仅是计算机专业本科生和研究生的一本很好的教材，对业界相关人士的研究开发工作也不无裨益。

本书的第 1、2 章及前言等由唐英翻译，第 3、4、5 章由毛家菊翻译，第 6、7、8 章由肖正光翻译，第 9、10、11 章由金恩华翻译，第 12 章由徐良贤翻译，附录程序中的注释部分由邱敏翻译。本书前半部分由金恩华初审，后半部分由唐英初审，并由唐英统排全书。最后，全书由徐良贤教授审校并统一给出了词汇表。

在翻译过程中我们尽量做到尊重原意、翻译准确，但由于水平有限，不当和疏漏之处在所难免，敬请广大读者不吝指正。

译者

上海交通大学电子信息学院

计算机科学与技术系

前 言

本书阐述了分布式计算的原理和实践，不仅适合学生学习，也可供从业人员及公司培训之用。在过去的十年中，计算机系统日益进步，大多数计算机都是连接在具有一致基础的某种网络之中的。小型企业中安装局域网越来越普遍，家庭安装的局域网也以越来越快的速度增加。软件技术必须跟上这一趋势，当前和未来的从业人员也是如此。按目前速度，所有计算机科研人员掌握分布式系统的工作原理只是个时间问题，因为大部分计算机及其应用都要用到这个技术。

本书读者

学习标准操作系统概念对计算机专业的大学生特别重要，同时，扩展分布式操作系统知识也是研究生和大学四年级学生以及业界工作人员迫切的和不断增长的要求。因此，很需要对分布式操作系统的原理、实际解决方案和方法进行研究。本书就能够满足学生和从业人员的这方面需要。

目标

本书从概念和实践的角度详细论述了分布式操作系统的各个主要方面，并且还包含了实际操作系统的相关范例，以加深概念和展示分布式系统设计人员所必须作出的决定。一些操作系统如 Amoeba、Clouds 和 Chorus（JavaOS 的技术基础）在全书中作为范例来讲解，此外，Windows 2000 案例研究还提供了一个实际商业解决方案的例子。针对分布式计算的各个方面，本书用 CORBA、DCOM、NFS、LDAP、X.500、Kerberos、RSA、DES、SSH 和 NTP 等技术说明实际的解决方案。附录中还有一个简单的 C/S 应用来演示关键的分布式计算程序设计概念，如 INET 套接字、pthread 和通过互斥操作实现同步。

总之，本书着重于分布式系统的原理、理论和实践。本书是为计算机从业人员、大学四年级学生和研究生编写的，并且假定读者已经学过基本的操作系统课程。我们希望这本书不仅对渴望充电的业界人员是有益的，而且对将本书作为教材来学习的学生来说也是有价值的。

内容组织和教学特点

本书分成两部分。从第 1 章到第 6 章为第一部分，提供分布式计算的基础知识，从第 7 章到第 11 章为第二部分，详述这些主题并更深入地研究高级分布式操作系统的主题。书中的教学特点如下：

1. 供进一步深入理解的详细说明。这些详细说明中包含诸如复杂算法和更深入的例子等内容。

2. 超过 150 幅的图表，以图解方式帮助阐明概念。

3. Windows 2000 案例研究，展示一个实际的商业解决方案。

4. 面向项目的习题（带有斜体数字），作为亲身体验。

5. 习题建立在前面几章的概念之上。

6. 参考资料来源包括：

- A. 供进一步学习的综述资料。
 - B. 研究论文。
 - C. 核心 web 站点和 ftp 站点。
7. 一个简化的分布式应用程序，以展示分布式程序设计的关键概念。
 8. 综合词汇表，集中给出主要的定义。
 9. 缩写字的完整列表，以帮助阅读并为快速查询提供一个集中存放地点。
 10. 各章小结。
 11. 完整的索引。
 12. 本书的 web 站点为 www.prenhall.com/galli。

对教师的建议

本书为教师提供了最大的灵活性，并具有一定的教学特点，以便教师能够根据班级需要和教学任务选择具体内容。在写本书时惟一的假定是读者已经学过了基础的操作系统导论课程。有些题材可能在操作系统导论课程中有所论述，但在本书中有时忽略或简单讲述，有的题材常常难于掌握或者可能已经淡忘，然而对分布式系统是很重要的，本书会在适当的地方插入这些内容。这些材料不必在课堂上讲解，但写进本书以保证学生在学习更高深的分布式题材时具有必备的基础知识。下面是一些建议，指导需要侧重实践的课程以及要求侧重研究的课程如何使用本书。需要侧重学习这两方面的研究生课程可以采用这两类建议。更多的信息可以在作者的 Prentice Hall 网站中获取，地址是 www.prenhall.com/galli。

侧重实践

下面是几个建议，适用于侧重实践的课程。

1. 给个别学生或一组学生一个或多个“项目练习”。这些练习在相关各章最后的练习题中用斜体数字指明。如果他们的设计和实现是在课堂上口述，则可以获得更多的实践经验。
2. 阅读有关实际实现的所有详细说明。
3. 在课堂上学习 Windows 2000 案例研究。
4. 建立一个个人或小组项目来研究 Windows 2000 的分布式特点。
5. 安排学生扩充或修改外科手术调度程序。可以简单地改变进程间通信的类型，也可以复杂地利用同样的分布式概念创建另一个程序。

侧重研究

下面是几个建议，适用于侧重研究的课程。

1. 让个别学生或一组学生就分布式操作系统的一个专题写一篇论文。每章后面的参考文献是较好的起点，这些练习可以包括一个口头讲解。
2. 展示一些从相关 RFC 中找到的讲授材料或者每章后面的研究论文。这些资料可以从 web 站点中获取，并将它们列在要求学生阅读的读物中。
3. 要求学生在网上查找相关的 RFC 文献和每章后面的论文，并作出摘要。
4. 在每章后面引用的参考论文中选择一部分，并创建一个活页簿以在整个课程中与此书进行同步学习。研究机构中的很多书店可为此提供所需的版权。

目 录

译者序		
前言		
第 1 章 分布式系统引论	1	
1.1 什么是操作系统	1	
1.2 什么是分布式系统	2	
1.2.1 流行的网络拓扑和特点	2	
1.2.2 ISO/OSI 参考模型	6	
1.2.3 分布式计算模型	8	
1.2.4 分布式与集中式解决方案	10	
1.2.5 网络与分布式操作系统	10	
1.3 什么是实时系统	11	
1.3.1 实时事件的特点	11	
1.3.2 影响分布式实时应用的网络特性	12	
1.4 什么是并行系统	13	
1.4.1 并行体系结构	13	
1.4.2 并行软件范例	16	
1.5 分布式应用实例	16	
1.6 小结	18	
1.7 参考文献	18	
习题	19	
第 2 章 内核	21	
2.1 内核类型	21	
2.2 进程和线程	22	
2.2.1 多线程进程介绍	24	
2.2.2 多线程进程范例	24	
2.2.3 多线程支持	25	
2.3 进程管理	26	
2.3.1 进程类型	27	
2.3.2 负荷分布和进程迁移	28	
2.4 进程调度	30	
2.4.1 识别用于调度的进程	30	
2.4.2 调度器的组织	32	
2.5 小结	33	
2.6 参考文献	33	
习题	34	
第 3 章 进程间通信	37	
3.1 选择因素	37	
3.2 消息传递	37	
3.2.1 阻塞原语	38	
3.2.2 非阻塞原语	40	
3.2.3 进程地址	40	
3.3 管道	42	
3.3.1 非命名管道	43	
3.3.2 命名管道	43	
3.4 套接字	44	
3.4.1 UNIX 套接字	45	
3.4.2 Java 对套接字的支持	48	
3.5 远程过程调用	50	
3.5.1 参数类型	50	
3.5.2 数据类型支持	50	
3.5.3 参数整理	50	
3.5.4 RPC 绑定	51	
3.5.5 RPC 认证	52	
3.5.6 RPC 调用语义	52	
3.5.7 SUN 的 ONC RPC	52	
3.6 小结	53	
3.7 参考文献	53	
习题	54	
第 4 章 内存管理	56	
4.1 集中式内存管理回顾	56	
4.1.1 虚拟内存	56	
4.1.2 页面和段	56	
4.1.3 页替换算法	58	
4.2 简单内存模型	59	
4.3 共享内存模型	59	
4.3.1 共享内存性能	60	
4.3.2 高速缓存一致性	61	
4.4 分布式共享内存	61	
4.4.1 分布式共享数据的方法	61	
4.4.2 DSM 性能问题	66	
4.5 内存迁移	66	
4.6 小结	68	
4.7 参考文献	69	

习题	69	6.5.4 DCOM 中支持的线程模型	95
第 5 章 并发控制	71	6.5.5 DCOM 的安全策略	96
5.1 互斥和临界区	71	6.6 CORBA 概述	97
5.2 信号量	72	6.6.1 CORBA 的 ORB	97
5.2.1 信号量的缺点	73	6.6.2 CORBA 的对象适配器	98
5.2.2 信号量评估	74	6.6.3 CORBA 的消息模型	100
5.3 管程	74	6.6.4 遵从 CORBA 标准	100
5.3.1 条件变量	74	6.6.5 CORBA 到 COM 的映射	100
5.3.2 管程评估	75	6.7 小结	100
5.4 锁	75	6.8 参考文献	101
5.4.1 轮转	76	习题	101
5.4.2 原子操作和硬件支持	77	第 7 章 分布式进程管理	102
5.5 软件锁控制	78	7.1 分布式调度算法选择	102
5.5.1 集中式锁管理器	78	7.1.1 调度层次	102
5.5.2 分布式锁管理器	79	7.1.2 负荷分布目标	103
5.6 令牌传递互斥	80	7.1.3 调度的有效目标	103
5.7 死锁	80	7.1.4 处理器绑定时间	104
5.7.1 防止死锁	81	7.2 调度算法的方法	106
5.7.2 避免死锁	82	7.2.1 使用点数方法	106
5.7.3 忽略死锁	82	7.2.2 图论方法	107
5.7.4 检测死锁	82	7.2.3 探查	109
5.8 小结	83	7.2.4 调度队列	110
5.9 参考文献	84	7.2.5 随机学习	111
习题	84	7.3 协调者选举	112
第 6 章 基于对象的操作系统	86	7.4 孤儿进程	114
6.1 对象介绍	86	7.4.1 孤儿进程清除	114
6.1.1 对象定义	86	7.4.2 子进程限额	116
6.1.2 对象的评价	87	7.4.3 进程版本号	116
6.2 Clouds 对象方法	88	7.5 小结	117
6.2.1 Clouds 的对象	88	7.6 参考文献	118
6.2.2 Clouds 的线程	89	习题	118
6.2.3 Clouds 内存存储	89	第 8 章 分布式文件系统	120
6.3 Chorus V3 和 COOL V2	90	8.1 分布式名字服务	120
6.3.1 基层:COOL 内存管理	90	8.1.1 文件类型	120
6.3.2 通用运行时系统层:COOL 对象	91	8.1.2 位置透明	121
6.3.3 特定语言运行时系统层	92	8.1.3 全局命名与名字透明	123
6.4 Amoeba	92	8.2 分布式文件服务	125
6.4.1 Amoeba 对象的标识和保护	92	8.2.1 文件多样性	126
6.4.2 Amoeba 的对象通信	92	8.2.2 文件修改通知	128
6.5 分布式组件对象模型	93	8.2.3 文件服务实现	128
6.5.1 标记	94	8.2.4 文件复制	129
6.5.2 远程方法调用	95	8.3 分布式目录服务	130
6.5.3 资源回收	95	8.3.1 目录结构	131

8.3.2 目录管理	131	10.2.2 物理时间的同步	160
8.3.3 目录操作	131	10.2.3 集中式物理时间服务	161
8.4 网络文件系统	132	10.2.4 分布式物理时间服务	163
8.4.1 NFS 文件服务	132	10.3 网络时间协议	164
8.4.2 NFS 目录服务	133	10.3.1 NTP 体系结构	164
8.4.3 NFS 名字服务	134	10.3.2 NTP 设计目标	165
8.5 X.500	134	10.3.3 NTP 同步模式	166
8.5.1 X.500 文件和名字服务:信息模型	135	10.3.4 简单网络时间协议	169
8.5.2 X.500 的目录服务:目录模型	135	10.4 逻辑时钟	169
8.6 小结	135	10.4.1 超前关系	169
8.7 参考文献	136	10.4.2 逻辑顺序	170
习题	137	10.4.3 带有逻辑时钟的总体排序	172
第 9 章 事务管理和一致性模型	139	10.5 小结	172
9.1 事务管理的动机	139	10.6 参考文献	173
9.1.1 更新遗失	139	习题	173
9.1.2 检索的不一致	140	第 11 章 分布式安全	175
9.2 事务的 ACID 特性	143	11.1 加密和数字签名	175
9.3 一致性模型	145	11.1.1 对称加密	176
9.3.1 严格一致性模型	145	11.1.2 非对称加密	179
9.3.2 顺序一致性模型	145	11.2 身份认证	183
9.3.3 偶然一致性模型	146	11.2.1 证书表	183
9.3.4 PRAM 一致性模型	147	11.2.2 集中式证书分送中心	186
9.3.5 处理器一致性模型	147	11.3 访问控制(防火墙)	189
9.3.6 弱一致性模型	148	11.3.1 包过滤网关	189
9.3.7 释放一致性模型	150	11.3.2 代理服务	190
9.3.8 懒释放一致性	151	11.3.3 防火墙体系结构	191
9.3.9 入口一致性模型	151	11.4 小结	192
9.4 两阶段提交协议	152	11.5 参考文献	192
9.4.1 准备提交阶段	153	习题	193
9.4.2 提交阶段	153	第 12 章 实例研究:Windows 2000	195
9.5 嵌套事务	154	12.1 概述:Windows 2000 设计	196
9.6 事务实现中的问题	156	12.2 内核模式综述	197
9.6.1 预读写	156	12.2.1 内核对象	199
9.6.2 中途退出的多米诺效应	156	12.2.2 硬件抽象层	200
9.6.3 保证恢复能力	156	12.2.3 设备驱动程序	200
9.7 小结	156	12.2.4 执行程序	200
9.8 参考文献	157	12.3 即插即用	201
习题	157	12.4 Windows 2000 中的 NT 文件系统	204
第 10 章 分布式同步	159	12.4.1 访问控制表	204
10.1 全局时间介绍	159	12.4.2 再解析点	205
10.2 物理时钟	159	12.4.3 存储管理	206
10.2.1 获得准确的物理时间	159	12.5 活动目录	206
		12.5.1 名字空间	208

12.5.2 通过修改日志实现复制和 可扩展性	209	12.8.2 加密文件系统	217
12.5.3 微软的索引服务器和 HTTP 支 持	210	12.8.3 微软安全支持提供者接口	218
12.6 微软管理控制台	212	12.9 HYDRA——一个瘦客户	219
12.7 集群服务	213	12.10 小结	220
12.7.1 集群服务概况	213	12.11 参考文献	220
12.7.2 集群抽象	213	习题	220
12.7.3 集群服务体系结构	214	附录 A 外科手术调度程序	222
12.7.4 为应用程序配置的集群服务	215	缩写词表	288
12.8 Windows 2000 安全性	215	术语表	292
12.8.1 安全配置编辑器	215	参考文献目录	303
		索引	318

第1章 分布式系统引论

计算机系统的进展从来没有像开发与创建分布式计算这样迅猛。因此对所有计算机研究人员深入理解系统的要求迅速增涨,这种一度限于对计算机专家的要求,已经在分布式计算技术中起着前所未有的作用 [Nev95]。在基本的集中式计算环境中,简单了解一下操作系统的概念就够了。但在分布式环境中的开发人员往往不仅要懂得分布式和实时概念,甚至并行系统,还要在更大的范围中实现它们 [SHFECB93]。个人经验显示,在高级环境中开发一个应用程序,开发团队常常要花费 50% 以上的时间来实现这些高级操作系统概念。

1.1 什么是操作系统

操作系统是计算机的任务管理者,它控制、管理、协调和调度所有的处理过程和相应资源。资源主要包括:

- ◆ CPU。
- ◆ 存储模块。
- ◆ 媒体存储器(磁盘驱动器、DVD[⊖]、CD-ROM 等)。
- ◆ 声卡和显卡。
- ◆ 总线和互连网络。
- ◆ 调制解调器和网卡。
- ◆ 显示器、终端、打印机和其他输出设备。
- ◆ 键盘、鼠标、扫描仪和其他输入设备。

操作系统的管理任务主要包括:

- ◆ 过程(或对象)管理。
- ◆ 通信。
- ◆ 存储访问/管理。
- ◆ 资源调度。
- ◆ 信息和资源安全。
- ◆ 数据完整性。
- ◆ 定时。

对于这些任务和部件,我们只希望系统能创建一个统一完整的视图或“图像”,传统上,操作系统的这些管理任务都由软件来负责,但在分布式系统中并不完全如此。我们讲解基本的操作系统概念,即使它们在分布式环境中是由硬件实现的。

为了形象地说明这个问题对于计算机科学家的重要性和复杂性,我们可以把操作系统看作音乐指挥。当然乐章越复杂,指挥的工作量和复杂性会迅速增加。如果同时指挥多个管弦

⊖ DVD 曾经是指数字影碟 (Digital Video Disk) 或者数字多功能影碟 (Digital Versatile Disk)。其格式已结合到现在的 DVD 中,现在的 DVD 已经不是一个字母缩写的组合了,虽然人们经常认为它是某种缩写。

乐队演奏不同的乐章，指挥的工作会变得更复杂。同时演出地点可能不同，并且还可能有时限制！很显然这是一个需全面解决的任务。但不管多复杂的任务我们都可以每次集中解决一个问题，一次克服一个障碍，并最终将揭示出全局的图像。在我们知道最终结果以前，整个图像会是清晰的，展现的是音乐般的美丽与和谐，而不是混乱、噪声或者杂乱无章的现象。作为第一步，让我们首先在 1.2 节到 1.4 节中分别定义分布式、实时以及并行计算的概念。

1.2 什么是分布式系统

分布式系统是一组异构计算机和处理器通过网络联接在一起，如图 1-1 所示。整组机器紧密配合工作，完成一个共同的目标。分布式操作系统的目标是为文件系统、名字空间、时间、安全和资源访问提供一个共同的、一致的全局视图。为了提供这个共同视图，所有成员系统都有许多限制和要求，因此分布式系统通常也称为紧耦合系统（每个规则总有例外！）。如果异构计算机系统通过网络互联起来，并且不是紧耦合的，那么通常称为**网络系统**。网络系统不提供共同的全局视图，也不对成员系统提出明显的要求。分布式系统或网络系统中的部件可以是简单的集中式系统部件，或者如 1.3 节所示的有实时限制的部件，甚至如 1.4 节所介绍的更复杂的并行系统。而且，分布式系统有时也会同时结合集中式的部件、实时性的部件和并行性的部件，在 1.5 节中可看到这样的实例。

由于 PC 机的每美元计算能力在迅速提高，网络和分布式系统都在不断普及。1.2.1 节介绍流行的网络；1.2.2 节提供一个由国际标准化组织（International Standards Organization, ISO）定义的开放式

系统互联（Open System Interconnection, OSI）参考模型；1.2.3 节介绍分布式计算模型；1.2.4 节讨论在一个分布式系统中决定用分布式还是集中式解决方案的有关问题；最后，1.2.5 节描述在分布式环境中可用的各种计算模型。

1.2.1 流行的网络拓扑和特点

虽然整本书和整个课程都是关于网络的，但本节只对网络简要介绍一下。希望这点最基本的介绍足以用来理解本书要学习的操作系统概念。

网络有两大基本类：局域网（LAN）和广域网（WAN）。典型的局域网由一个单位所拥有，范围只有几公里。尽管许多现代广域网（比如 ATM 网）已经大大降低了错误率，但在 20 世纪 90 年代中期之前，局域网的错误率往往比广域网低一千倍。一个节点可能有几个子网或更小的局域网，大的局域网可以由多个小局域网组成。小局域网可能以如下方式连接，

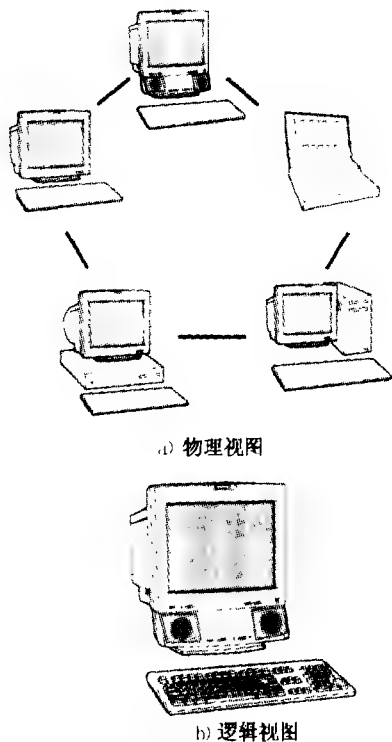


图 1-1 网络环境中的计算机

其算法可参见详细说明 1.1。

详细说明 1.1

连接局域网子网的路由器算法

令 p1 和 p2 为设备的两个端口。

令 OUTPUT (a, b) 将端口 b 接到的消息内容输出到端口 a 所连的网络。

令 DESTINATION (a, b) 为真, 当且仅当端口 a 接到的消息必须使用端口 b 来送达目的地。

令 DIF_PROTOCOL (a, b) 为真, 当且仅当端口 a 和端口 b 使用不同的网络协议并且 DESTINATION (a, b) 为真。

令 CONVERT (a, b) 将端口 a 接到的消息转换为使用端口 b 所用协议的消息, 并发送到端口 b。

转发器算法:

While ()

{

OUTPUT (P1, P2);

//P2 接到的所有消息都输出到 P1

OUTPUT (P2, P1);

}

网桥算法:

While ()

{

if DESTINATION (P1, P2);

Then OUTPUT (P1, P2);

//仅在到达目的地需要经过的情况下转发

if DESTINATION (P2, P1);

Then OUTPUT (P2, P1);

}

路由器算法:

While ()

{

If DIF_PROTOCOL (P1, P2);

//仅在到达目的地需要经过的情况下转发

Then CONVERT (P1, P2);

//只在必须的情况下转换

Else If DESTINATION (P1, P2);

Then OUTPUT (P1, P2);

//仅在到达目的地需要经过的情况下转发

If DIF_PROTOCOL (P2, P1);

Then CONVERT (P2, P1);

Else If DESTINATION (P1, P2);

Then OUTPUT (P1, P2);

}

1. **转发器**：一个非智能型的设备，只是简单地将一个网络里的东西全部转发到另一个网络中去。这两个网络必须使用相同的协议，也就是说它们是同类型的网络。

2. **网桥**：一个智能设备，只将一个子网里的数据转发到目标子网，或者到达目标所需要经过的子网。这些网络必须实现相同的网络协议。

3. **路由器**：比网桥更先进之处在于它能连接使用不同协议的局域网网段。路由器能同时连接两个以上的网络。

4. **主干**：不包含用户的局域网，它只连接其他网络而不连接用户。如图 1-2 所示。

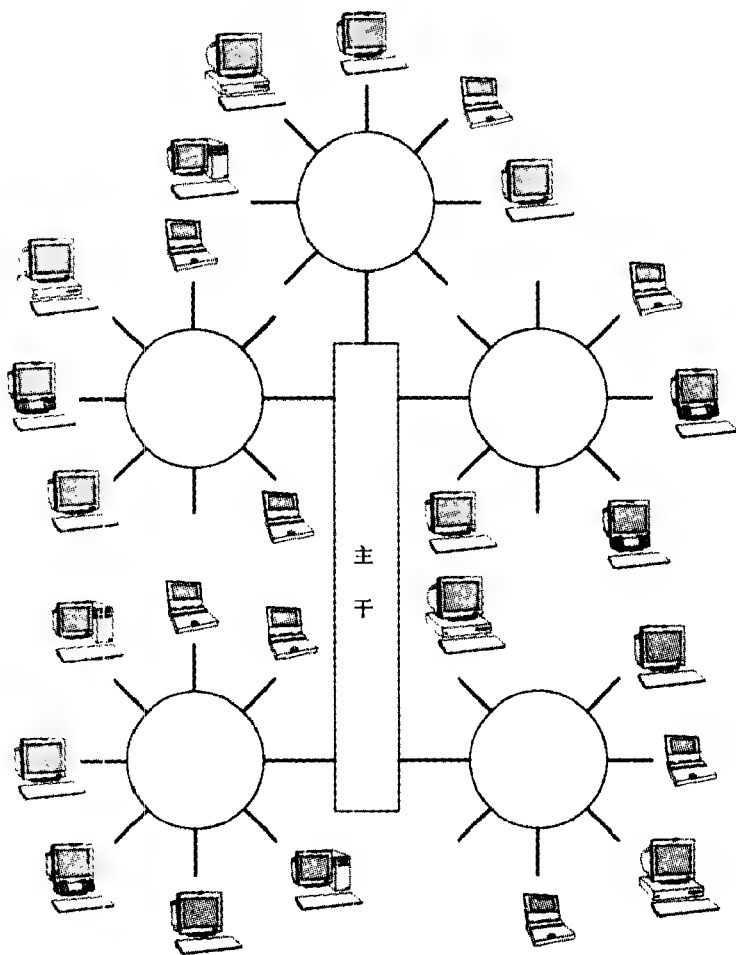


图 1-2 通过主干来连接多个局域网子网

局域网可以有线的或无线的。有线局域网通过物理线缆连接，无线局域网通过无形的通信通道如红外线或无线电连接。有线局域网有三种常用拓扑，如图 1-3 所示。其中最常用的两种拓扑是以太网（在 IEEE802.3 标准中的正式命名是 CSMA/CD [IEEE85a]）和令牌环网（在 IEEE802.5 标准中指明 [IEEE85b]）。由于这些拓扑提供的操作服务类型根本不同，它们之间没有直接竞争。以太网类似于将邮票贴在信封上，而令牌环网类似于次日发送并要求回执。基本以太网运行在 10Mbps 上，运行在 100Mbps 上的快速以太网是当前流行的版本，而千兆以太网则运行在 1000Mbps 上。以太网以尽可能快的速度发送信息，但不确认信息是否

被正确地接收。而且标准以太网协议不支持信息优先权，也不保证传输时间。对大多数应用来说，它在达到容量的 50% 之前运行得很好，之后它开始迅速变慢，并且几乎所有的 CS-MA/CD 网络在超过约 60% 的容量时将崩溃而无法传送任何信息。

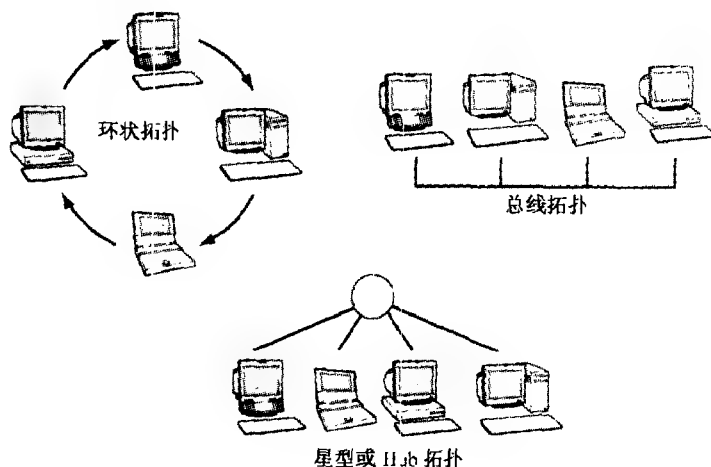


图 1-3 常用有线局域网拓扑结构

令牌环模拟带回执的次日传送，能确认所有信息被正确送达。它允许优先级并保证传输时间，这使它对实时应用特别有吸引力。与以太网不同，令牌环网在高负荷时也不会崩溃，而是继续顺利地发送信息。这个协议是考虑优先权的公平访问。令牌环网依所用种类不同，运行在 4 到 16Mbps 之上，跟 10 Base^① 以太网同一个档次，但实现成本是以太网的两倍。

光纤分布式数据接口（Fiber Distributed Data Interface, FDDI）是另一种常用网络协议，经常用于主干或高速局域网。FDDI 操作在 100Mbps，其实现是建立在两个反相令牌环网上。这个协议设计支持同步信息流，因此对实时应用很有吸引力。

无线局域网由于不断增长的移动需求和有线局域网固定的重配置费用而变得越来越普遍。无线局域网通常有一个较高的初装成本，但维护成本较低。无线局域网可以是基于无线电，也可以是基于红外线。基于无线电的局域网当前可达 4Mbps，并且容易安装，因为无线电信号能穿过不透明的物体传播，但这也意味着网络信号传到建筑物之外。因此基于无线电的局域网有时不如有线局域网或红外线局域网安全。不过，由于传播频谱技术的利用，信号可以以噪音的方式传播或者令信号不断跳频传播，这样无线电局域网也可以更加安全。此外，有线局域网所采用的各种安全手段如第 11 章的加密技术，也可以用于无线电局域网。红外线局域网如今可达 10Mbps，但红外线信号不能穿过不透明物体。这一点使红外线局域网很难安装，因为发送器和接收器必须在一条直线上并且相互可见。而且红外线信号可能受到光噪音的干扰，影响了可靠性。

许多分布式系统并不是在局域网，而是在广域网。要将一个局域网连接到一个广域网，需要用到网关。信息发送到网络上在传输之前被分成更小的片，称做帧或包。有两个常用的方法可以用来在广域网上发送这些包。第一个方法是电路交换，用于公用交换电话网（public-switched telephone network, PSTN）。这类网络在进行任何数据传输之前提供一个固定的数据

① 10Base 以太网一般指运行在 10Mbps 下的以太网网络。

速率通道，这个保留的通道完全用于当前的数据传输。这个方法需要大量的设置时间并降低了网络吞吐量，但能保证一个信息在其传输过程中只有传输延迟。第二个方法是包交换。每个信息被分成称为包的小单元，包的大小有固定的上限。它们可以使用一个虚拟电路以面向连接的方式发送，也可以使用数据报以无连接的方式发送。虚拟电路要求同一个信息的每个包都沿着同样的路径按正常顺序传输，这通常要求先发送一个设置包。相反，数据报对相关包的传输路径和次序不作限制，在目的地把收到的数据报依正常次序重新排列，并再组装成原始信息。这个方法可靠性较弱，但更健壮和高效。

广域网常用的网络协议包括帧中继和异步传输模式（asynchronous transfer mode, ATM）。帧中继允许网络消息包含拥塞控制和速率信息，以便在网络过于繁忙时丢掉该消息。ATM 网络以速度和支持多媒体文件传输而著称，尽管这些特点并非 ATM 网络所独有。ATM 网中的每个包尺寸相同，都是 53 字节。ATM 网把这些 53 字节的包称作单元（cell）。通过使用不同的帧格式，ATM 适应层（ATM adaptation layer, AAL）支持固定位速率和可变位速率传输，以及面向连接和无连接传输。它在设计时考虑了支持实时多媒体应用。当然，像 Internet 这样的广域网使用了所有网络类型的组合。

1.2.2 ISO/OSI 参考模型

ISO/OSI 参考模型 [ISO84] 描述了在各种网络模型中所执行的所有计算任务的一种可能划分。这个参考模型如图 1-4 所示划分成七层，层数越低越接近网络。第一到第三层依赖网

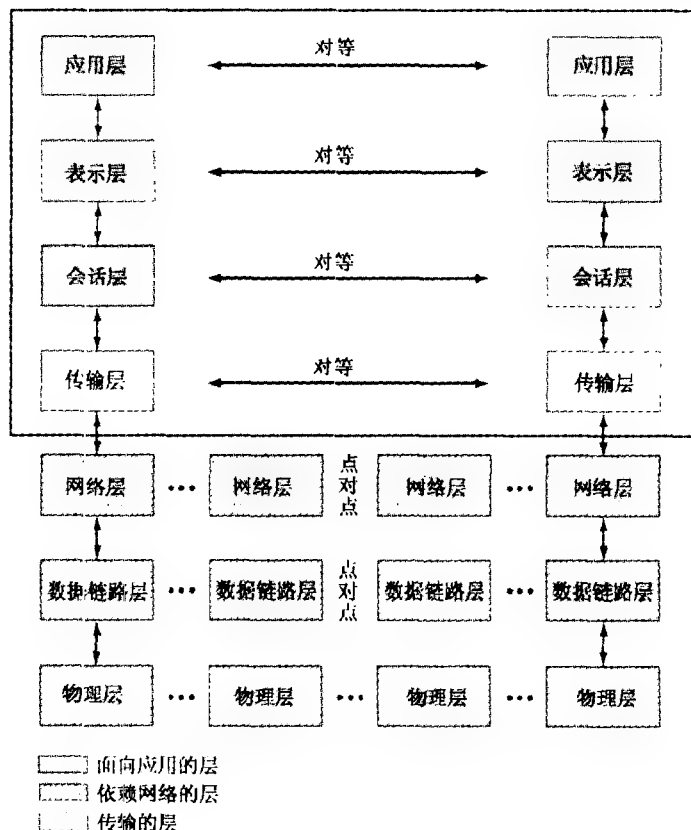


图 1-4 ISO/OSI 参考模型

络，与跟系统直接面对的具体网络协议密切相关。第五到第七层是面向应用的，第四层是介于网络和应用之间的传输层。操作系统概念始于第四层的接口，包括了第五至第七层。这个参考模型与 Internet 所用的传输控制协议/互联网协议 (TCP/IP) 的协议族关系参见详细说明 1.2。现在我们简要定义和描述 ISO/OSI 参考模型的七个层次。

详细说明 1.2

TCP/IP 协议族与 ISO/OSI 参考模型

TCP/IP 协议族是一组设计用来访问异构网络或互联网的协议。TCP/IP 最初是 20 世纪 60 年代为政府投资的高级军事研究项目网 (ARPANET) 而设计的 (此时 ISO/OSI 还没有出现)，直到 1978 年才演进成今天这样的在互联网上普遍采用的形式 (ARPANET 最初的协议是网络控制程序——NCP)。当互联网 (internet) 术语的首字符写成大写的 I 时，专指因特网 (Internet)。1983 年第一个包含了 TCP/IP 的 UNIX 版本发布，为家用 PC 机环境提供的第一个包含了 TCP/IP 的 Windows 版本 (Windows95) 在 1995 年发布。TCP/IP 分成四层。图 1-5 显示了这四层与 ISO/OSI 参考模型的关系。下面描述这四个层。

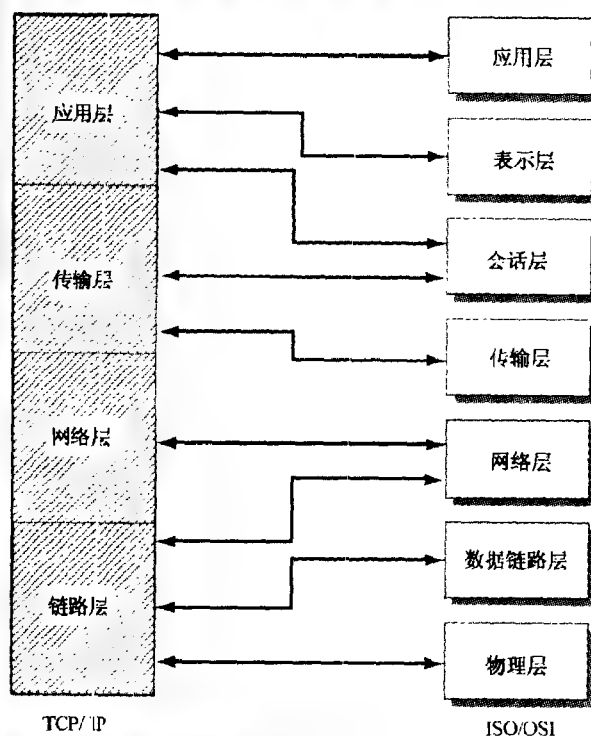


图 1-5 TCP/IP 与 ISO/OSI 参考模型的关系

1. 链路层：链路层是最低的层，包含了 ISO/OSI 参考模型最低的两层。从字面上看，这意味着网卡和该卡所用的操作系统设备驱动器在这一层中。

2. 网络层：网络层是倒数第二层，直接映射到 ISO/OSI 参考模型中的网络层。这一层包括网际协议 (IP)、网际控制报文协议 (ICMP) 和网际组管理协议 (IGMP)。

3. 传输层：对应于 ISO/OSI 参考模型的传输层。该层有传输控制协议 (TCP) 和用户数据报协议 (UDP)。TCP 提供包括确认机制的可靠传输，是一个高级类别的服务。另一方面 UDP 只管发送包，由上面的层负责解决可靠性问题。作为一种低级类别的服务，UDP 以高效著称，常为实时应用所倚重。在这种情况下，应用程序必须自己负责实现所需要的可靠性。

4. 应用层：应用层对应于 ISO/OSI 参考模型最高的三个层次。该协议中的服务包括远程会话的 *telnet*，文件传输协议 (FTP)，使用简单邮件传输协议 (SMTP) 的 *e-mail*，名字服务器协议 (NSP)，以及简单网络管理协议 (SNMP)。

1. 物理层。物理层是最低的层，负责向线路上发送位。所有协议都关注一个适配器上有多少个连接器。电气工程师对此层尤感兴趣。

2. 数据链路层。对于局域网而言，数据链路层分成两个子层：媒体访问控制和逻辑链路控制。逻辑链路控制不是局域网所独有的，它使用协议来确认和重发帧，以此作为一种错误控制方法。数据链路层也负责流控制，控制网络上信息传输的数量和速度。因此流控制可帮助防止网络及其资源的“过载”。由于局域网是一种共享的广播媒体，因此我们必须使用协议来达到有效的和高效的利用。常用的协议包括 CSMA/CD (以太网) 和令牌环协议，如 1.2.1 节中所述。

3. 网络层。网络层主要功能是为一条信息从源地到目的地进行路由。由于局域网上无须路由，因而该层在局域网环境下是空的，不提供任何功能。

4. 传输层。传输层的首要任务是判定在网络上通信所必需的服务类别。最高一类的服务被认为是面向连接的，并提供完全的错误控制和确认服务。最低一类的服务是无连接的，在该层上不提供这样的错误恢复服务。

5. 会话层。该层控制源地和目的地在整个通信期间的相关组织工作和同步协调。

6. 表示层。该层负责消息数据的语法，并执行相应的转换。它包括在 ASCII 码和 EBCDIC 码之间的转换，如压缩和加密等。

7. 应用层。应用层的主要功能是提供用户界面，建立授权，以及资源的管理和使用。

ISO/OSI 参考模型分层概念的另一个重要特点是点对点层与对等层。点对点层中的功能在源地和目的地以及源地和目的地之间的每个中间节点执行。因为可能有上千个节点处于源地和目的地之间，消息从源地向目的地传输的过程中，点对点的功能就可能需要执行上千次。第一层到第三层中定义的所有功能都是点对点的。对等层（第四层到第七层）中的功能只在源地和目的地执行，总共两次。作为一个点对点的功能的例子，让我们来看看检查信封地址的功能。这个功能在投寄局执行，也在收信局执行，还在投寄局和收信局之间的所有中转局执行。而且在目的地收信局和邮递员都要看信封地址。最终，目的地执行检查地址的功能以确定实际接收人。相反，阅读信件内容的功能则（希望！）只由发信人和收信人来做，因此这是一个对等功能的例子。

1.2.3 分布式计算模型

有一句老话说：“厨师多了搞坏汤。”因此分布式计算的重要问题是如何管理所有这些厨师，以免搞坏了汤。分布式计算有两个主要方法：C/S 模型和对等模型。C/S 模型因简洁而

非常流行；对等模型是更高级的模式，正随时间的推移而越发普及。

C/S 模型

C/S 模型很象企业中的经理/下属模型。客户是经理，而服务器是下属。客户想做某件事，并把这个任务交给服务器。服务器去完成这个任务，并通知它的老板，也就是客户。客户向服务器提交请求，服务器回应客户，如图 1-6 所示。这个操作发生在 ISO/OSI 模型的会话层。它通过进程间通信来实现，相关内容将在第 4 章讲述。在 C/S 模型中，可以有几个客户向几个服务器提交请求。同样，某个服务器可以满足多个客户的请求。在本地环境，有两个基本方法来实现 C/S 模型：工作站模型和处理器池模型。

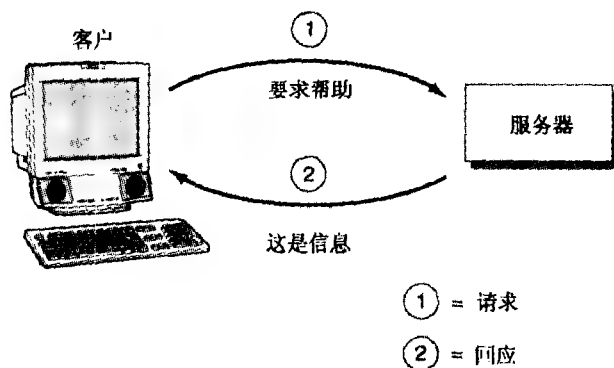


图 1-6 C/S 模型

工作站模型允许每个本地用户有一个小型工作站，这个工作站本身没有足够计算能力来完成用户要求的任务。当需要更多能力时，用户的工作站通过网络发出一个请求，希望找到另一个空闲的工作站以协同工作。这种模型本身带来一系列问题，相关讨论在第 7 章进行。

在处理器池模型中，只给每个用户一个简单的终端，所有处理能力都集中在一处，就像大型机。需要资源时，用户向服务器发出一个请求。服务器控制所有的处理能力和服务，以及这些能力和服务的分布。这方面的讨论也在第 7 章进行。

对等模型

对等模型是 C/S 模型的自然演进。它类似于一群同事在一起工作，计算组的每个成员可以向其他成员提交服务请求。因此 C/S 模型可当作是对等模式的简化和前身。一些应用要求使用对等模型，以提供多对多关系。如 [BCEF94] 中所指出，分布式系统可能需要一个大型数据库来存储所有设备和主机的信息，这就需要管理部件来管理和优化使用数据库的系统。同样，数据库系统也需要管理部件来优化分布式请求。这里，哪个是客户，哪个是服务器？都不是。大家要像在对等环境中的同事那样一起工作，如图 1-7 所示。注意：一个环境

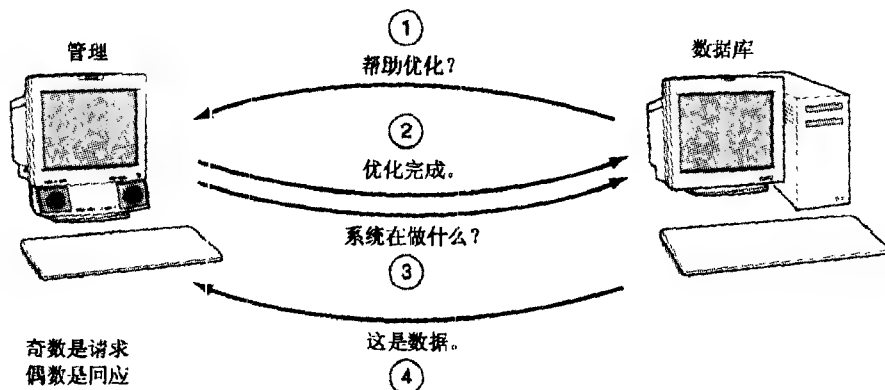


图 1-7 对等模型

如果支持对等协作，也并不排除 C/S 模型，因为有些工作并不需要团队工作方法。支持对等模型的是一种被称为平衡配置的方法。

对等架构的例子有实现 ISO/OSI 数据链路层中逻辑链路控制子层的高层数据链路控制协议。这个协议在异步平衡方式上操作时支持对等模型，并在一个被称为 X.25 的标准中实现了。IBM 的高级对等网络（APPN）也支持对等计算。由于这些架构的进一步细节超出了操作系统的范围，我们不再在此详细讨论，感兴趣的读者可以在本书中找到更多关于 X.25、高层数据链路控制和 APPN 的信息，如 1-7 节中所列出的一些参考资料。

1.2.4 分布式与集中式解决方案

分布式环境中，多种任务的解决可以选择分布式或集中式解决方案。集中式解决方案把整个决策及与决策有关的各种信息都放在一个位置上。这类解决方案是最简单的解决，也是改编非分布式系统算法的最容易方法。但这里有几个严重缺点：首先，中央管理单元就变成关键单元，一旦关键单元崩溃，整个分布式系统也就失效了；第二，中央管理单元周围的网络流量将增长，因为系统中所有成员都不得不跟这个位置通信。但从积极的一面看，集中式架构中应用软件的升级倒是变得相当简单了，因为只有一个系统需要新软件。

相对来说，分布式解决方案虽不受关键部位之累，但也有其缺点。分布式解决方案常常增大了整个网络中的流量，因为许多分布式解决方案都需要使用广播信息。此外，让多个点的信息保持一致也是很困难的。一些点由于网络延迟，收到的数据更新或软件更新的信息可能比其他点晚。由于信息不完整，优化解决就不可能。而且，分布式解决方案通常要求各种成员之间协作。

1.2.5 网络与分布式操作系统

区别网络操作系统和分布式操作系统的主要因素是它们的透明度和对成员系统的要求。

系统要求

分布式系统要求所有成员都分别运行它们自己拥有的同一分布式操作系统的拷贝，但分布式操作系统也常在本地另一种不同的底层操作系统或集中式操作系统之上运行。运行于一个底层操作系统之上的分布式操作系统常称为中间件（middleware）。网络操作系统不要求成员运行同样的系统，只要求它们在拷贝文件和远程登录之类操作上使用同样的协议。因此网络操作系统和分布式操作系统之间的最大区别，就是网络操作系统不隐藏在不同系统上操作这一事实。也就是说，它并不将远程操作对用户透明，也不支持如下任何一种透明性。

透明性

我们现在解释分布式操作系统支持的各种透明性，其中一些透明性比另一些透明性是我们更希望获得的。

- ◆ **名字透明性**：资源的名字并不能表明文件、数据或进程在分布式系统中的物理位置。如果用户改变位置，他们看到的系统视图也不会改变，因为名字跟实际位置没有关联。总之，这意味着某种全局命名方案。
- ◆ **位置透明性**：当提供位置透明性时，用户不知道他们所使用的资源的实际物理位置。这意味着支持名字透明性和访问透明性，在涉及文件和处理器时是有用的，利用它们时没有附加成本。但是对于打印机这类资源，通常不希望有位置透明性。当用户打印

一个文档时, 需要知道它将在哪儿打印出来。

15

- ◆ **访问透明性**: 访问透明性比位置透明性更进一步, 用户不能区分本地资源和远程资源。在界面、命令和时间(最困难的方面)上, 对资源的本地访问和远程访问是完全一样的。由于时间的约束, 这通常只能在本地(在同一个局域网中)分布式系统上获得。访问透明性最困难的方面之一是支持安全访问, 而不受访问透明性的妨碍, 如第11章中所述。
- ◆ **迁移透明性**: 迁移透明性意味着用户不能觉察到一个资源或他们的工作是否被迁移到分布式系统中的另一个位置。要实现这一点, 名字透明性是必要的前提。这个问题将在第4章和第7章详细讨论。
- ◆ **复制透明性**: 复制透明性允许在整个系统中存在文件和服务器的多份拷贝。而且这些拷贝对用户透明, 并且所有的修改和更新对多个拷贝一致。关于复制透明性的更多知识请参见第8章和第9章。
- ◆ **并发和并行透明性**: 多个进程可以利用同一个资源, 或者一个进程可以同时利用多个资源, 而不会冲突或知晓。这些任务被自动和高效处理以充分利用分布式系统。这方面的内容将在第10章讨论。
- ◆ **故障透明性**: 如果分布式系统中的一条链路或系统发生故障, 整个系统不会停止运行。故障透明性是分布式计算相对于集中式计算的重要优势, 后者只能任由你的一台, 也是惟一的一台计算机崩溃而毫无办法。这个问题将在第9章详细讨论。

1.3 什么是实时系统

实时系统必须满足有限响应时间的限制, 否则将承受严重后果。如果这个后果是性能的降低而不是故障, 则系统被称为**软实时系统**。如果后果是系统故障, 则这个系统被称为**硬实时系统**。在硬实时系统中, 系统故障可能导致死亡或其他威胁生命的后果。如果能容忍小概率的系统故障, 则这种系统被称为**稳固实时系统**。通过网络实现的多媒体应用就是稳固实时系统的例子。你可能已经听说大多数计算机系统都必须满足某些时间限制, 否则将承受后果, 事实的确如此。大多数系统是软实时系统。在本书中谈到实时系统时我们通常指硬实时系统和稳固实时系统, 它们可能引起严重的后果。实时计算的典型教学工具是用计算机控制火车, 如果程序失败则列车倾覆。如果这不是练习的话, 将有人人员伤亡。这是一个我们都希望尽量做得完美的领域。传统的实时大量用在电信、航空和国防工业领域。

16

有两类实时系统:**反应式**和**嵌入式**。反应式实时系统包含一个与环境不断交互的系统。这个交互可能是领航员不断按控制键, 或一个人不断按键或按钮。嵌入式系统是用以控制一个安装在更大系统中的特定硬件。现实中的例子就是汽车中控制油气混合的微处理器。

关于实时系统的更多知识将在下面两小节中讲解。实时事件特点将在1.3.1节中讨论, 影响分布式实时系统的网络特点将在1.3.2节中说明。

1.3.1 实时事件的特点

实时事件有如下三类:**异步的**、**同步的**和**等时的**。异步事件是完全不可预测的, 比如暴风雨中闪电将击中的确切位置。一般来说, 异步事件是由外部原因引起的。例如电话呼叫是异步的, 电信程序不知道用户什么时候会用电话与系统连接。相反, 同步事件是可以预测

的，如果它要出现的话，会按精确的规则出现。同步事件通常是由内部原因引起的，比如嵌入式部件。如果事件有任何类型的传播延迟，例如网络连接，则该事件不是同步的。如果事件严格出现在一个给定的时间范围内，它可以被称为**等时的**（isochronous，读做 eye-sock-run-us）。但这一类在技术上作为异步事件的一个子类，并不总被应用所适应。等时事件的一个例子是在分布式多媒体应用中接收音频字节。音频字节必须在这个应用正常工作所可能接受的时间范围内接收到。可以预期这些字节流不会正好在预定的时刻接收到，因为可能由于网络错误而遇到网络延迟。

值得指出的是这些术语在网络科学范围内也有意义。具体来说，异步描述了一个以非连续方式独立传输每个数据包的网络或网络协议。但是，同步网络或网络协议则是把一条消息作为一个连续的位流发送。

1.3.2 影响分布式实时应用的网络特性

下面是底层网络中能影响分布式实时应用的四个因素。

1. 网络响应时间。
2. 带宽/成本。
3. 路由优化。
4. 微网络特性。

第一个因素是基本网络响应时间不能忽略。响应时间是由于数据从源地传播到目的地所产生的延迟。在集中式系统，系统内通信的响应时间是可以忽略不计的。但由于基本的物理限制，分布式系统的响应时间则成问题，尤其是对电信这样的分布式实时系统。例如，即使以光的速度传播，信息横越美国大陆也要 20ms。用高速公路作类比，这就是假定没有红灯（在各个节点没有延迟）、拥塞或交通事故（数据错误引起重传）。如果应用要求在远远散布在各地的几个部分之间大量通信，可能无法达到实时的要求。而且，网络延迟使得所有分布式事件成为异步的（或者等时的），精确预测时间就不可能了。

第二个因素是带宽/成本的权衡。分布式实时应用要求固定带宽。当前一条在东京与大阪之间的 155Mbps 的连接（比如一个 OC3 连接）每个月要花费约五十万美元。因此在技术上也许能实现播送实时视频，但却不经济。好在带宽每年都在不断降价，虽然目前还不能确定海量带宽何时普及到大众并真正结束“全球等待”（world wide wait, www）的时代。历史已经一再向人们显示，不管有多少资源可用（不管资源是计算机物理资源还是金钱），我们总是能找到办法来消耗这些资源，并且梦想有更多资源。现在很难想早期只有 16K 字节内存的卫星如何工作，以及 300 波特家用调制解调器曾经如何值得炫耀（不是炫耀它在计算机博物馆内的价值）。

第三个因素是分布式实时系统必须驻留在网络中，因此受制于网络的拓扑结构和网络对分布和路由的影响。理论上说，网络设计要使源地到目的地之间经过的节点数最少，但这个理论在实践中并不现实（参见习题 1.8）。打个比方，设想你在交通繁忙时刻从市中心驾车前往郊区，地理上最短的路径其实并不就是花费时间最短的路径（也就是最快的路径）。地理最短的路径可能由于种种原因而速度很慢，包括速度限制或者说路径的数据传输速率、路径上的流量、通路数或者说路径的容量。如果该城市有收费公路，最短的路径可能不是最便宜的路径。所有这些问题在信息高速公路上也存在，因此分布式实时应用的整体性能总的来

说受当前的基础设施约束。

第四个因素是分布式实时系统受微网络特性的影响,比如网络交换机的缓存大小、队列大小,以及网络所允许通过的包大小。理论上,这些特性应该对上层和应用透明,但实际上由于编写软件时程序员的技术水平限制和缺少支持标准,因此常出现问题。我的一个同事曾遇到过一个未能充分利用高速 WAN 而且是不能用的视频流程序(匿名的),是由于视频程序员编写了 65K 字节大小的 UDP 包。这说明了应用受限于微网络特性,应用的这些方面应由更低级软件支配。没人知道一个软件会使用多少年(如 1999 年 9 月 9 日^①和 2000 年软件危机),以及哪一类技术将使用它(比如视频流程序)。代码越灵活,应用的可用性就越强。

18

1.4 什么是并行系统

像分布式系统一样,并行系统包含多个处理器。但是这多个处理器(通常)是同构的,并且位于同一台计算机中。除了多个处理器外,并行计算机可以设计成每个处理器都有自己的本地内存。因此并行计算机也称为**多计算机**(multicomputer)。如果并行的处理器共享内存,它们就叫做**多处理器**(multiprocessor)。这些多处理器常按访问共享内存的开销来分类。如果所有处理器的访问时间一样,则多处理器的内存称为一致内存访问(Uniform Memory Access, UMA),否则称为非一致内存访问(Non-Uniform Memory Access, NUMA)。不共享内存的多处理器(每个处理器有自己的本地内存)也叫做**非远程内存访问**(NO Remote Memory Access, NORMA)多处理器。1.4.1 节讲述一些体系结构,用于将众多处理器组织成单台计算机。1.4.2 节讲述并行计算环境中所使用的一些软件范例。由于并行计算机通常用来解决最复杂的问题,并且远比单处理器计算机昂贵,因此也常称之为超级计算机。并行操作系统的任务就是有效利用所有资源来解决那些复杂问题,以及提高计算速度。

1.4.1 并行体系结构

我们现在讲述一些常见的并行体系结构。详细说明 1.3 描述了一个引自 [Fly72] 的流行体系结构分类法。用来连接各个处理器的互联网络大致决定一个并行系统的体系结构。如果并行系统是一个多处理器,互联网络也用来连接处理器和共享内存模块,如图 1-8 所示。互联网络可能设计成允许同步或异步通信。处理器和内存模块之间的数据和信息必须有路由控

19

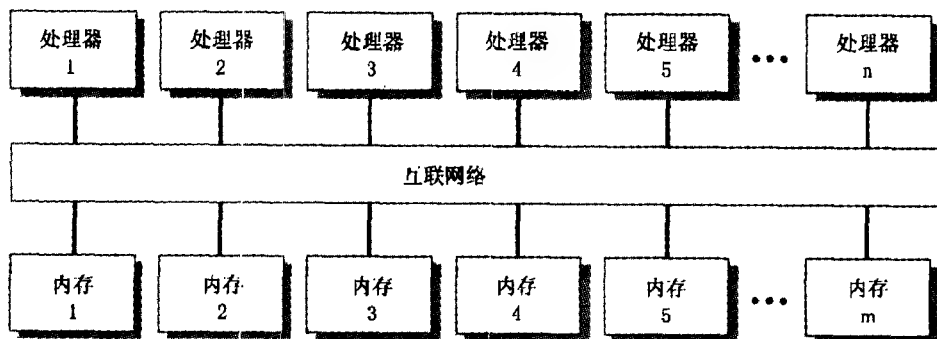


图 1-8 多处理器中的互联网络

① 1999 年 9 月 9 日会表示成 9999, 这个数字被一些程序员在一些应用中作为例外处理, 从而当时钟走到此日期时将造成混淆, 除非能预知此问题。幸好, 在许多地方这个问题都将在千年虫测试过程中被辨认出来。

详细说明 1.3

Flynn 的体系结构分类

Flynn 根据如何处理机器指令和数据来分类各种体系结构。有如下四类 [Fly72]。

SISD

第一类体系结构是单指令流单数据流 (SISD)。这样的计算机遵循传统的冯·诺依曼模型，其例子包括传统的单处理器计算机，比如简单的 PC 机或 20 世纪 80 年代早期的主机。

SIMD

第二类体系结构是单指令流多数据流 (SIMD)。这类体系结构适合于对大批数据执行同样的指令。例如对一个矩阵的第 i 行数据乘以一个值 v 。这里单个指令是乘，但作用于该行的每个元素。有一种并行体系结构模型叫做向量-数组处理器，比如 CRAY 超级计算机就是 SIMD 体系结构。向量机将多个（典型为几千个）处理器排列成栅格形式。向量-数组处理器擅长计算的数据类型是向量。其他 SIMD 例子还包括 Iliac IV 和连接机 CM-2。不幸的是 SIMD 体系结构的并行系统发展速度快于操作系统技术，因此 SIMD 系统通常利用一个使用传统单处理器操作系统的前端计算机来控制。这样 SIMD 计算机就被当作“前端”计算机的一个设备了。

MISD

第三类体系结构是多指令流单数据流 (MISD)。Flynn 为了完整性而列入了这个类，但目前没有这一类系统。

MIMD

最后是多指令流和多数据流体系结构 (MIMD)。分布式和并行系统都可归为 MIMD 体系结构，但本节我们只讨论并行机。MIMD 体系结构利用多个处理器取得数据和指令，然后异步执行。MIMD 的例子包括 *Sequent*、超立方体机和晶片机。

制。有些互连网络支持分布式路由控制，另一些使用集中式路由控制。此外，互连网络的基本拓扑可以是静态或动态的。动态拓扑允许互连网络在运行期间重新配置，通过使用交换机来实现各种连接。当一个互连网络能实现每一个可能的连接时，它就被称为拥有 100% 的置换能力。现在来看看互连网络的各种拓扑。

总线

最简单的互连网络是总线。

总线是静态的，并且不要求任何类型的路由。每个处理器和内存模块都连接到总线上，数据的所有事务和移动都发生在这个共享的总线上，如图 1-9 所示。*Sequent* 并行计算机用一条总线作为互连网络。虽然这是一个比较简单的互连网络拓扑，但总线限制了模块间的带宽，从而产生了

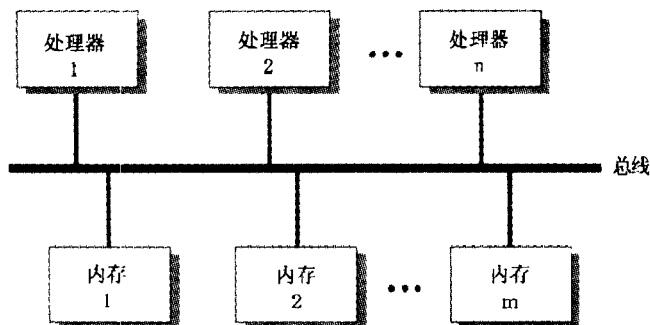


图 1-9 用做互连网络的总线

通信瓶颈。

交叉互联网络

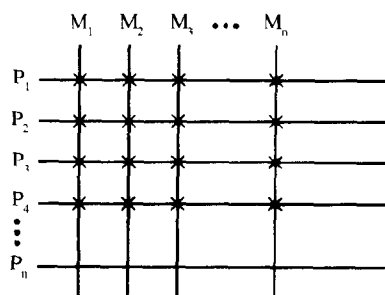
允许动态配置的最简单互联网络是交叉网络。交叉网络是连接 n 个内存模块和 n 个处理器的 $n \times n$ 连接。这样的网络要求 $O(n^2)$ 个开关, 如图 1-10 所示。这么大的开关意味着它是一个昂贵的互联网络。这个动态配置具有 100% 的置换能力, 并且多个处理器可以同时使用网络, 只要没有使用同一个交叉开关。

超立方体

一个著名的静态互联网络是超立方体。超立方体是按其维数定义的。具体来说, k -立方体是一个有 2^k 个开关的网络。由于开关数目是 2 的指数, 因此这些立方体有时候也称为二进制超立方体。开关编号为 0 到 $2^k - 1$, 如图 1-11 所示。网络中的两个开关当且仅当它们的二进制编号只有一个位不同时连接。因此开关 0100 跟开关 0110 之间有连接, 但开关 0100 跟开关 0111 之间没有连接。这种拓扑的优点是两个开关之间的距离容易计算。在这种拓扑中, 单个处理单元位于每个开关中。

洗牌交换互联网络

最具动态性的互联网络之一是洗牌交换网络。网络中开关允许动态重配置, 并有四种操作方式, 如图 1-12 所示。其中控制位决定了开关盒的操作方式。这类互联网络的取名源自其行为方式很像重新洗牌。具体来说, 理想的洗牌是实现高位目的地与低位目的地完全交叉访问, 如图 1-13 所示。多级洗牌交换互联网



X = 开关
P = 处理器
M = 内存模块

图 1-10 交叉互联网络

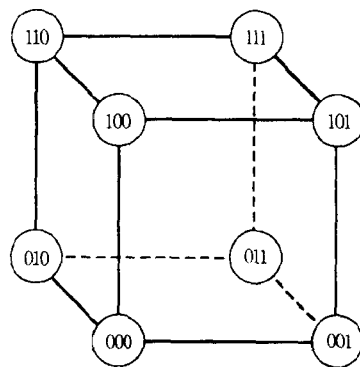


图 1-11 3 次超立方体的互联网络

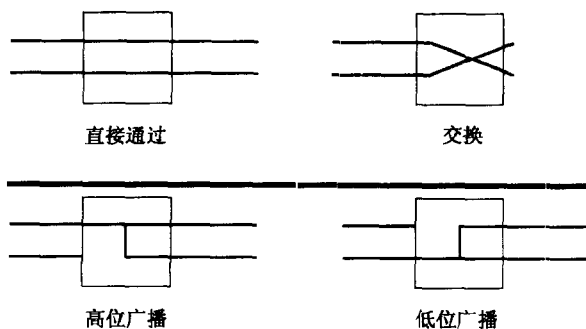


图 1-12 洗牌交换互联网络中的 2×2 开关盒

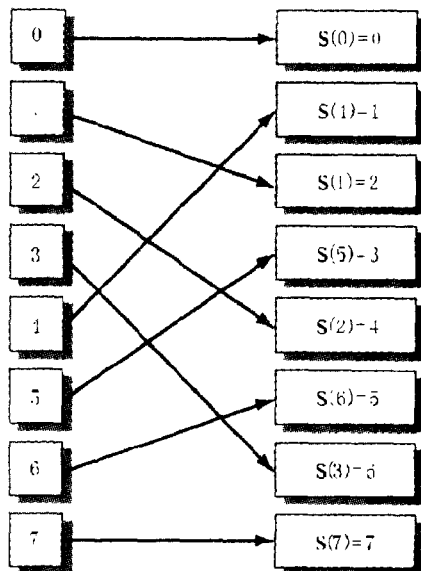


图 1-13 理想的洗牌

络则如图 1-14 所示。在这个例子中使用了八个输入。由于任何源地都能根据其地址定位目的地，因此在这个网络中寻址是很简单的。具体来说，最高位的二进制位决定在网络第一级的动作，第二高位决定第二级，依此类推。二进制位 1 表示通过底部输出，而二进制为 0 则要求通过顶端输入。其他基于洗牌交换网络的互联网络包括 Benes、Banyan、Omega 和 Theta 互联网络。

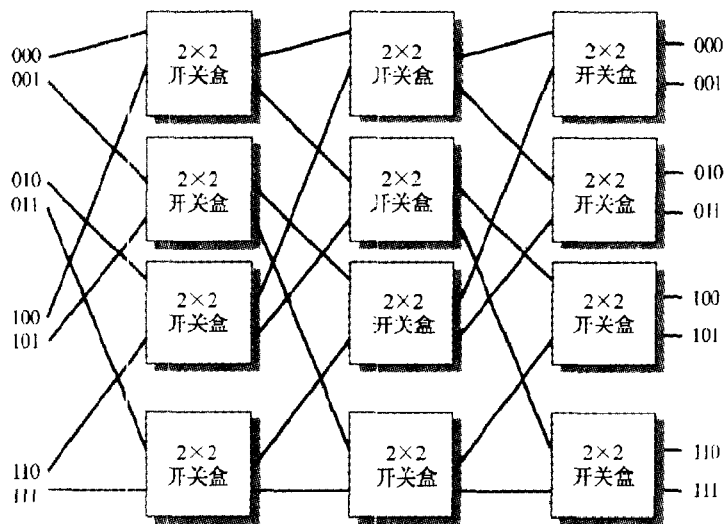


图 1-14 多级洗牌交换互联网络

1.4.2 并行软件范例

并行系统在其多个处理器间分配任务有三类基本方式。分配工作可由程序员或系统来实现，也可能由其体系结构决定。

1. 复制代码，分割数据。在这种情形下，每个处理器被分配相同的任务，各自处理一批不同的数据。例如每个处理器得到一个分类程序，但其中一个处理器要按标题和日期来分类所有 IEEE 期刊，而另一个处理器要按标题和日期分类所有 ACM 期刊。

2. 流水线。在这种情形下，每个服务器分配的工作依赖于前一个处理器完成的任务，很像一条流水线。整个任务要到最后一个处理器完成其工作时才最终完成。这个方法是仿照亨利·福特汽车制造生产线的思路。整个工作分成不同的处理阶段，每个处理器分别负责完成一个阶段，并送往下一个处理器。

3. 树状结构。这种情形类似于写算法时的自顶向下设计方法。第一个处理器有一个任务要完成，而这个任务由几个子任务组成。然后将这些子任务分给其他处理器去做，而这些处理器又进一步将这些子任务再分成几个子任务交给别的处理器去做，依此类推。

1.5 分布式应用实例

分布式系统不但包含传统的集中式机器部件，而且也有实时部件和并行部件。图 1-15 描述电信应用领域中一个这样的应用。虽然这里没有给出任何公司生产中的任何一个特定的系统架构，但是给出了在这样一个系统中我们一般能找到什么。现在我们来解释每个部件。

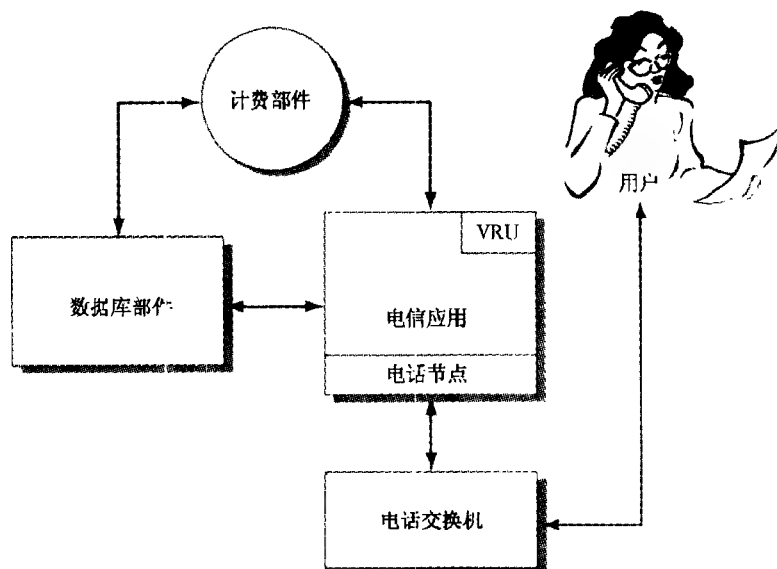


图 1-15 分布式系统的例子

24
25

图中用户代表希望访问该电信系统的多种应用程序用户。当用户拨入时，他们通过电话交换机访问电话应用，交换机是根据接收到的他们在话机上拨出的号码决定的。用户希望即时和连续的服务，使得这一分布式系统成为具有严格时间限制的分布式实时系统。严格要求系统具有实时性并不仅仅是为了让顾客满意，还考虑到电话服务还连接了顾客与 911 报警、医生办公室、毒品控制中心，以及其他事关生命安全的实时性场所。

电话交换机能同时处理多个呼叫，这是因为它是一个包含多条接入和接出电话线的并行硬件部件，对多个同时呼叫是同时操作处理的。交换机要求每年停机时间少于 1 分钟，因此必须具有极高的容错性，并内置冗余部件。交换机是一个典型的嵌入式实时部件的例子。

电话节点是带有电话卡的计算机的通用术语，比如能使计算机接收传真的传真卡，或者使计算机能处理电话会议的会议卡。电话节点只运行于基于电信的应用。在商业化环境中，可能有一组电话节点运行应用软件。这些电话节点可能作为电信系统的一个分布式子系统，用完全分布式的方式组织。由于冗余或容错性能，这种部件也常作为并行部件，因为电信应用不允许停机（用户也不能允许！）。1.4.2 节中描述的每个软件范例都可用于这个部件。电话节点最不常见的用途是复制代码和分割数据，因为这样就要求每个电话节点都拥有所有硬件卡的复本，从而极大地增加了系统的硬件成本。

电话节点部件一般用作语音应答单元（voice response unit, VRU），或者与 VRU 交互，VRU 根据用户电话拨号情况向用户播放预先录制的信息。一个预录信息的例子是“对不起，电话不通，请挂机并重拨。”VRU 也常提供交互式语音应答（interactive voice response, IVR）能力，比如上面那个例子后面可以再加一句“如果您需要帮助，请按‘0’。”当用户输入 0 时，将转接给服务人员。这里 IVR 就是一个交互式实时部件的例子。

计费部件是一个实时应用，根据用户的活动来计算相应的费用。某些情况下，计费部件还要不断检查用户花费了多少，并与其许可的额度比较。例如，如果用的是预付费电话卡，

应用程序要确保用户使用的金额不超过卡里的数值。这里就需要与数据库部件交互。如果这个部件不精确，电信公司就要蒙受损失。

26

数据库部件可与应用的许多方面相关。通常电信公司为大量用户服务，因此要求数据库完全分布以分散负荷，减少用户的等待时间。而且数据库部件内的子部件可能是并行结构，以加快速度。这个部件必须处理本书几章中讨论到的问题，包括第4章提到的分布式内存，第5章提到的并发控制，第8章提到的分布式文件，以及第9章提到的事务管理。

虽然还有许多细节需要讲解，但显然即使从简单角度来看电信应用，它也要求有各种系统类型的部件，包括实时的和并行的。电信业只是复杂分布式应用最典型的应用领域之一，其实计算机行业的每个方面都存在这种系统，并且还在不断创建。

1.6 小结

分布式系统是计算机行业和日常生活中不断增长的应用领域，通常包括实时和并行部件。每个部件都向操作系统提出各自的一些待处理的问题。并行和分布式系统都有多个处理器。并行系统包含紧耦合部件，而通常的分布式系统拥有大量不同的部件。网络系统并不试图隐藏使用多个远程资源的事实。因此，虽然分布式系统可能使用远程处理器而不被用户发觉，而网络操作系统的用户则必须输入所有的远程指令。实时系统本身就具有严格的时间限制。尤其这些限制本身具有挑战性，在处理分布式实时系统的网络延迟时更加如此。

在本书中，分布式计算的基本内容在第一部分中讲述。第一部分包含一些在入门课程中偶然提及但对学习分布式系统很重要的材料。第二部分的材料建立在第一部分的题材之上，是对这些基本问题的扩展，主要集中在分布式系统中更深入、更高级、更难理解的问题细节。第12章中的 Windows 2000 实例对研究单个系统如何解决分布式操作系统中的许多问题进行了全面的考察。通过这些学习，希望能令读者举一反三，踏入分布式计算的世界，创建并看到一个浑然一体的系统，尽管它是由单独的、不同的部件组成。

1.7 参考文献

27

下列参考文献包含关于分布式、实时和并行系统以及网络 and 并行计算机的更多资料：[Akl97, Ber92, Bla96, DDK94, Hal96, Hun95, Kop97, Kri89, Lap92, Sin97, SSF97, Sta97, Ste94, Tan95 和 Tan96]。与本章内容有关的一些传统研究论文包括 [BCEF94, BrJo94 (论文集), CaMu94 (论文集), Mul93a (论文集), SHFECB93 和 Son95 (论文集)]。

现在我们列出有关高级操作系统的一些 Internet 上的主要站点，其中讨论的题材对本章和全书的学习有益。IEEE 计算机协会实时系统技术委员会的主页拥有与实时技术有关的大量链接列表，其地址为 <http://cs-www.bu.edu/pub/ieee-rt/Home.html>。关于超大规模集成电路 (VLSI) 和并行体系结构方面论文的一组链接可在 <http://rtlab.kaist.ac.kr:80/~yunju/wavelet-papers.html> 上找到。Internet 并行计算文档可在 <http://www.hensa.ac.uk/parallel/> 上找到。宾夕法尼亚大学分布式系统技术文档和研究资料库包含大量研究论文，可在 <http://www.cis.upenn.edu/~dsl/library.html> 上找到。一批计算机科学书目，可在 <http://linwww.ira.uka.de/bibliography/index.html> 上找到，其中操作系统有 13 000 条，并行处理有 32 000 条，分布式计算有 28 000 条，还有超过 22 000 个技术报告链接。网络 CS 技术报告库提供全球各大学和实验室关于计算机科学丰富的技术报告，地址为 <http://www.ncstrl.org>。WWW 虚

拟图书馆的计算类目也很有帮助,地址为 <http://src.doc.ic.ac.uk/bySubject/Computing/overview.html>, 相关缩略词的虚拟实体 (Virtual Entity of Relevant Acronyms, VERA) 位于 <http://userpage.fu-berlin.de/~oheia/bdd/veramain-e.cgi>。IEEE 的总主页在 <http://www.computer.org/>, 计算机协会 (ACM) 的主页 <http://www.acm.org/> 对其发行的相关资料提供了很有用的搜索引擎, 甚至包括一些摘要。计算机在线研究知识库 (CORR) 地址为 <http://www.acm.org/corr/>。操作系统专题组 (SIGOPS) 的学报可在 ACM 的数字图书馆里找到, 其地址为 <http://www.acm.org/pubs/contents/proceedings/ops/>。

习题

- 1.1 为什么今天的计算机科学家必须懂得高级操作系统的原理?
- 1.2 网络操作系统和分布式操作系统之间的主要不同之处是什么?
- 1.3 哪些局域网和广域网协议更适合实时应用?
- 1.4 为什么无线局域网会普及?
- 1.5 为什么说用分布式操作系统来执行 ISO/OSI 对等的各层是有利的?
- 1.6 对等分布式计算模型比 C/S 模型好在哪里?
- 1.7 本章讲述了七个分布透明性。请对每一种透明性提出并描述一个应用, 说明在分布式系统中这种透明性对它有什么好处?
- 1.8 Traceroute 是一个工具, 显示网络中从源点到目的地所经过的实际路由。在你的本地机器上或者在一个基于 Web 的 traceroute 网关上 (如雅虎的 <http://net.yahoo.com/cgi-bin/trace.sh>), 或者在 <http://www.tracert.com> 的多重 traceroute, 或者用关键字 traceroute 在 Internet 上搜索到的其他站点上, 运行 traceroute。在每天三个不同的时间执行并记录从你所在大学 (或本地的 Internet 节点) 到选择的五个不同目的地之间的路由跟踪结果, 并连续进行一个星期。注意每次应该跟踪同一批源点和目的地。
 - a. 每次得到的结果一样吗?
 - b. 每次的路线都是最短的吗? 为什么?
 - c. 对于分布式实时应用采取优化路由, 这意味着什么?
- 1.9 为如下各种系统举出一个实例。
 - a. 软实时系统。
 - b. 稳固实时系统。
 - c. 硬实时系统。
 - d. 反应式实时系统。
 - e. 嵌入式实时系统。
- 1.10 为下面各种事件举出一个例子 (不要使用书中已有的例子)
 - a. 同步事件。
 - b. 异步事件。
 - c. 等时事件。
- 1.11 说明多计算机和多处理器之间的不同。
- 1.12 对下面各种网络, 列出它们用于互联的并行体系结构时潜在的一个优缺点。

- a. 总线。
 - b. 交叉互联网络。
 - c. 超立方体。
 - d. 洗牌交换互联网络。
- 1.13 为下列各个并行软件范例举出一个潜在的应用。
- a. 复制代码，分割数据。
 - b. 流水线。
 - c. 树状结构。



第2章 内核

操作系统中影响最大的部分之一是内核。实际上，内核是计算机的厨师长，确保所有的厨师都弄不坏汤！内核是操作系统中的特权部分，对于在它完全控制之下的所有系统的所有资源都具有完全的访问特权（分布式系统内核一般不控制每个成员，可能只控制本地系统），而其他程序就没有这个特权。内核控制进程管理，包括进程迁移（2.3节）、进程调度（2.4节）、内存管理和地址空间（这两部分的内容见第4章）。这些功能在一个被保护和隔离的地址空间中操作，以防止被其他应用程序意外或恶意地冲突和操纵。我们从2.1节讲解两个基本内核类型开始，2.2节介绍与进程有关的概念，而内核的各种任务和解决方法则在2.3和2.4节讨论。

2.1 内核类型

根据内核大小及所包含的内容，内核设计可分为两大类。第一类是 UNIX、OS/360、VMS 所用的单片集成内核。正如名字所示，单内核系统拥有大型内核，包含大多数的操作系统服务，包括进程、内存、文件、命名和设备管理，以及进程间通信。换句话说，单内核就是一个完整的操作系统。第4章和第5章讨论的系统级共享内存或基于消息的通信，也常用于进程间通信以实现它们的功能。由于内核中没有保护边界，安全性的缺乏，以及单内核的尺寸和复杂性，使得调试、验证和修改都很困难，因而招致了批评。单内核不仅仅难以调试，而且根本就不存在调试支持。在分布式环境中，不是每个拷贝或每个成员都需要系统的所有功能，这就使得硬件与内核的接近成为分布式系统中的问题。因此分布式环境更倾向于采用第二种内核类型——分层内核。

31

分层内核着重于可分模块的设计。分层内核最常见的一类是微内核，其目的是使内核尽可能的小。分布式实时系统一般使用微内核。实际上，在嵌入式实时系统中微内核就是整个

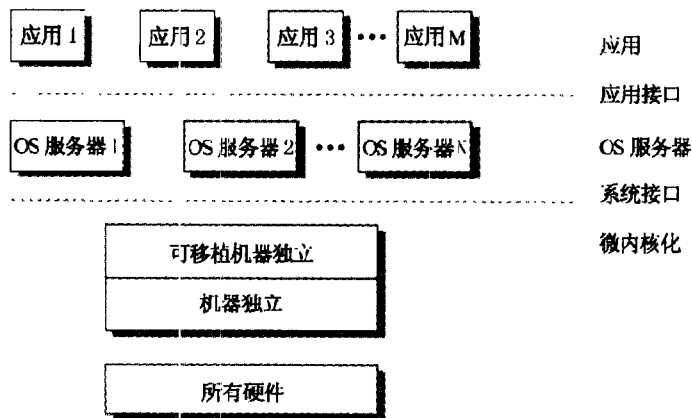


图 2-1 微内核设计

101601
1018108

操作系统。这些内核只提供最少的服务，比如进程间通信和有限的设备、进程、线程和内存管理。其他服务是在内核之上实现，作为用户空间里的操作系统服务。这些服务依赖于有效的进程间通信机制。微内核倾向于使用模块，从而易于设计、实现、安装、扩展和重配置。模块设计的一部分是使所有依赖体系的部分位于同一处，如图 2-1 所示。这有助于提高可移植性。这也使微内核吸引分布式操作系统，如第 6 章所描述的 Chorus (JavaOS 的基础技术) 和第 7 章所描述的 Mach，以及第 12 章 Windows2000 (基于 NT 技术) 实例研究中描述的 Windows NT。

32

2.2 进程和线程

执行的程序常常被抽象地称为进程。从系统的角度来看，与进程具体对应的概念就是工作。进程的思想来自集中式系统时代，但这个概念在高级系统中变得更难以定义。复杂性在分布式或并行环境中，单个进程可以使用多个控制线程来利用扩展的可用资源。不考虑操作环境和运行什么应用的话，所有事项都可以归结为运行进程。进程可能需要执行几个计算。一些环境中，特别是分布式和并行环境中，允许每个单独的计算作为一个独立的线程来执行，以便利用多个可用处理器。当一个操作系统允许一个以上的执行线程时，称为多线程的。2.2.1 节介绍多线程进程，多线程的组织在 2.2.2 节中讨论，2.2.3 节则讨论多线程进程的各种支持。详细说明 2.1 提供多线程编程的 POSIX 语言支持，常用 C 和 C++ 来实现。详细说明 2.2 提供支持多线程编程的 Java。此外，附录中的外科手术调度程序也是为在多线程环境中操作而编制的。

详细说明 2.1

POSIX 对多线程编程的支持

IEEE 协会在八十年代组织了一个特别任务组叫 POSIX，目的是为操作系统接口制定一系列标准。特别是 POSIX.1c 标准规定了多线程编程接口。虽然大多数人把 POSIX 标准看作面向 UNIX 的，但也有很多非 UNIX 的、符合 POSIX 标准的操作系统，包括 IBM 的 OS/2，微软的 Windows-NT 和 DEC (数字设备公司) 的 VMS。在 POSIX 中线程是一个可选特性，并且几乎总是能在 UNIX 及其变体中见到，但一般很少在其他环境中出现。作为一个标准，使用其线程应该能使采纳该标准的所有应用都变得更易移植。

所有使用 POSIX 线程的程序必须包含 pthread.h 库。编写多线程代码时通常使用基本模板程序，并修改以符合自己的要求。另外，大多数多线程程序要求并发控制，这是第 5 章的主题。虽然整本书都是关于线程编程的，但还有很多细节未包括在本书内，我们探讨的是管理和使用线程所必须的四个主要功能。2.6 节提供了关于多线程编程的一些参考文献。在附录中有一个多线程应用的例子——外科手术调度程序，它使用了多线程并用 C 语言写成。

作为一个标准，返回值 0 表示成功的函数调用。

```
int pthread_create(pthread_t* tid,
    const pthread_attr_t *attr,
    void* ((*start)(void*), void* arg));
```

tid 参数包含被创建线程的进程 ID。如果属性参数 attr 为空, 线程将拥有默认属性。最后一个参数是该线程将开始执行的函数名。

```
int pthread_exit (void* status);
```

函数 exit 用于终止分离的线程。如果线程不是分离的, 它就是可连接的。这样的线程使用一个带有相同参数的、叫做 pthread_detach 的函数来终止。status 参数是在线程终止后需要继续存在的变量的地址, 这个变量含有终止线程的退出状态代码。因此它不能是线程的局部变量。

```
int pthread_join (pthread_t tid, void** statusp);
```

函数 pthread_join 等待 pthread_t 退出, 然后将它的完成状态存储在 statusp 所指向的位置。

```
int pthread_self
```

这个函数返回一个给定线程的进程 ID。注意: pthread_create 函数也会返回这个 ID。

详细说明 2.2

JAVA 对多线程编程的支持

作为语言的一个重要组成部分, Java 对线程提供了丰富的支持。当一个对象作为线程创建时, 它必须跟别的对象一样对待。有两个支持线程的类: Thread 和 ThreadGroup。Thread 用以创建独立的线程, 而 ThreadGroup 则用以创建一组线程。对该组的操作作为一个整体来执行。系统级的程序通常使用 ThreadGroup。

Java 的 Thread 类包含七个构造函数。

```
public Thread();
public Thread(Runnable target);
public Thread(ThreadGroup group, Runnable target);
public Thread(String name);
public Thread(ThreadGroup group, String name);
public Thread(Runnable target, String name);
public Thread(ThreadGroup group, Runnable target,
               String name);
```

无论怎样实例化一个线程, 它都必须传递给一个可运行的对象。在 Java 中, 一个可运行的对象就是一个能定义 run () 方法的对象。如果不将线程实例化为一个可运行的对象, 就必须给出该线程的子类, 然后让子类定义自己的 run () 方法。

Thread 类中包含的其他一些方法如下:

```
public final void join () throws InterruptedException;
public synchronized void start () throws
    IllegalArgumentException;
//启动一个线程
public final void stop ();
//终止一个线程
```



```

public final void suspend ();
    //挂起一个线程
public final void resume ();
    //重启一个挂起的线程

```

2.2.1 多线程进程介绍

在传统的操作系统中，每个进程使用单个地址空间独立运行，并使用单个控制线程。在分布式和并行环境中，进程允许运行多个执行线程。这些线程有时也称为轻量级进程。每个线程有自己的程序计数器和堆栈，但与兄弟线程共享进程的地址空间和全局变量，如图 2-2 所示。在所有非并行环境，以及所有不实现共享内存的并行环境中，线程也共享同一个 CPU。同一个父进程的所有线程在共享地址空间时没有保护分界（如果使用普通线程概念而又不能接受没有内存地址空间保护，最好使用真正的轻量级进程来代替线程）。每个执行线程可以用准并行的方式运行，从而可以通过利用额外的可用资源来提高吞吐量。像进程一样，线程也可以派生子线程。由于创建和终止线程要耗费时间，因此许多应用程序就创建线程池，在创建新线程或子线程时重用线程。现在我们来解释组织多线程进程的各种方法。

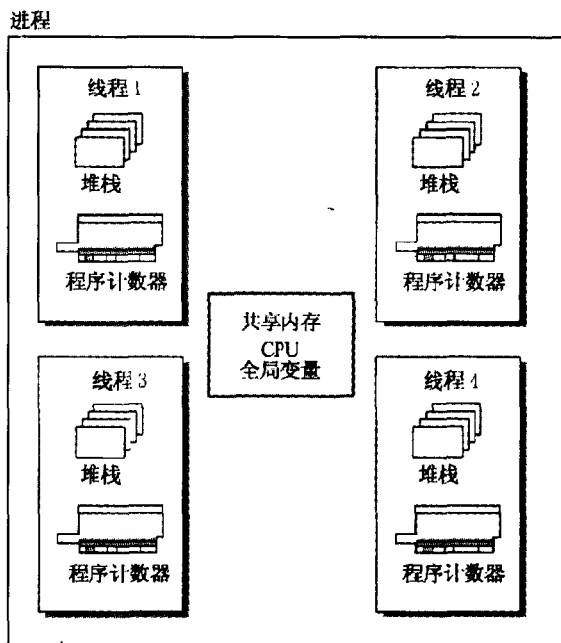


图 2-2 多线程进程

2.2.2 多线程进程范例

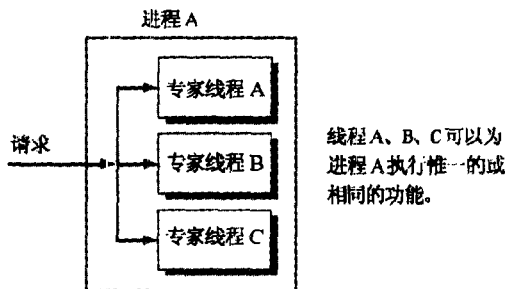
这里给出构造同一父进程的多个线程的三个常见范例，如图 2-3 所示。

1. 专家范例。在专家范例中，所有的线程平等，但为父进程执行自己的专用服务。换句话说，所有的线程能够但不要求执行惟一的专用服务，对每个线程的服务请求直接由父进程触发。

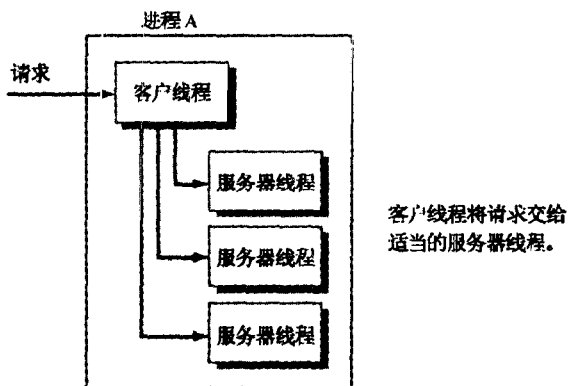
2. C/S 范例。在 C/S 范例中，客户接收从父进程或线程来的所有请求，并把请求转给一个适当的服务器线程。这种结构是基于分布式进程的 C/S 范例。

3. 流水线范例。在流水线范例中，一个线程的输出是下一个线程的输入，工作原理很像流水线。第一个线程的输入是进程的输入，最后一个线程的输出是进程的输出。这种结构是基于流水线并行软件范例。

A. 专家范例



B. C/S 范例



C. 流水线范例

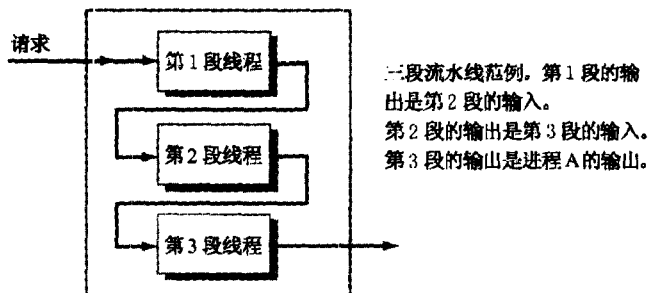


图 2-3 多线程的进程范例

2.2.3 多线程支持

内核可以支持线程，或者系统可以对线程提供用户级支持。当内核支持线程时，内核在其保护内存空间中管理一个线程状态信息表。固定尺寸的状态表维护所有线程的状态，每个线程一个项，包含的几个字段表示线程的状态、优先级、寄存器值和其他有关信息。当一个线程阻塞时，内核使用一个单级的调度器来选择运行另一个线程。这个新线程可能与原线程有关，也可能无关而来自另一个进程。

当线程在用户空间中实现时，它们运行在管理线程的运行环境的顶端。因此，运行环境系统维护状态信息表。线程的完整功能由运行环境控制，包括调度，从而使用户能够精确控制，比如选择运行哪个线程等。用户级支持使得支持多线程程序成为可能，哪怕所用的底层

操作系统不支持运行多个线程。

我们现在比较线程的这两种实现。状态表的位置影响线程使用的灵活性和可扩展性。由于内核级线程实现方法使用固定尺寸的状态表，因此它的灵活性不强，更重要的是可扩展性也不强。调度是另一个重要的问题。支持内核级线程就强制程序员使用内核实现方式；相反，如果线程是在内核之上的用户级实现，程序员就能控制是选择执行一个兄弟线程还是另一个进程的线程。如果速度很重要，就必须考虑到内核级线程的切换意味着使用（缓慢的）内核陷阱。

考虑到这些问题，人们可能倾向于使用用户级线程。但是，它们也有一些缺点。具体地说，内核之上的任何进程，包括支持用户级线程的进程，并不具有内核那样的控制能力。不同于内核实现方式，如果一个线程控制了 CPU，用户级实现方式不能强迫线程放弃自己的控制地位，让给其他线程去运行。即使创建一个协议来使线程请求一定量的执行时间，用户级实现方法也不能像内核那样强制协议。最后，用户级实现方法还要注意是否有线程执行了导致线程被内核阻塞的命令，如果有则整个进程（如运行用户级线程的进程）也被阻塞。一个常用方法是使用护套例程来捕获线程执行的每个系统调用。护套例程首先检查以确保执行的线程不会阻塞用户级线程管理进程，再将它交给系统。还必须记住，任何一个看上去无害的命令都可能引起一个页错误，从而导致整个线程包被阻塞。第 13 章中的实例研究描述了 OSF 的 DCE（开放式软件基金会的分布式计算环境）实现用户级线程。如果运行在一个有内核级支持的系统之上，DCE 也能利用内核级线程实现方法。调度问题将在 2.4 节进一步讨论。

2.3 进程管理

进程管理是用来控制进程及其所有组件。进程管理涉及进程及如下所有相关组件的管理：

- ◆ 地址空间目录。
- ◆ 寄存器。
- ◆ 程序计数器。
- ◆ 堆栈指针。
- ◆ 系统调用状态。
- ◆ 所有相关的执行线程。
- ◆ 所有的进程文件及其状态（打开或关闭）。

当进程主动或被动地放弃一个处理器的控制权后，整个状态必须切换出来并进行保存。进程的状态包括进程的所有相关部件的当前状态（如前面所列）。如果进程只是短暂阻塞，所付出的成本可能远超其值。在这种情况下，进程可能就在阻塞等待期间空转或紧循环。不幸的是，系统很难确定一个进程要等待多久，但还可以用试探法。其中一些选择包括总是空转或从不空转。如果你从不空转，系统可能执行上下文切换，消耗掉比等待所需时间更多的处理时间。如果你总是空转，并且进程阻塞的时间很长，就会浪费很多处理时间。这个控制可由系统使用，也可由父进程使用。进程管理包括下列责任：

- ◆ 创建一个进程。
- ◆ 识别进程类型（在 2.3.1 节中讨论）。

- ◆ 进程迁移（将一个进程的执行位置转移到另一个处理器或主机，在 2.3.2 节中讨论）。
- ◆ 调度（什么时候运行进程，在 2.4 节中讨论）。
- ◆ 查询进程的状态。
- ◆ 进程状态报告（例如进程已崩溃，进程已退出等）。

如图 2-4 所示，进程可能处于三个状态之一：阻塞、运行和等待（可运行但当前没有执行）。进程阻塞可能是因为等待输入，访问一个资源，或者在跟另一个协作进程同步。进程在就绪状态可能因为它是一个新进程，它刚刚取消阻塞，或者它使用了最大的时间片并不得

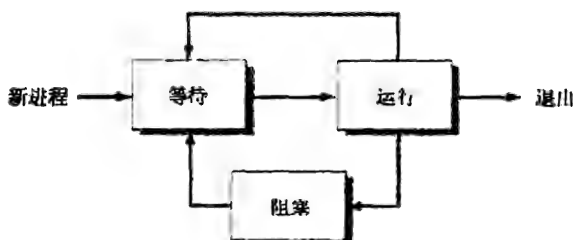


图 2-4 进程状态

得不被转出。详细说明 2.3 描述了 Amoeba 的进程管理系统。最后，描述进程管理的更深入知识，包括分布式和实时进程调度，将在第 7 章中讲解。

39

详细说明 2.3

Amoeba 进程管理

Amoeba 是一个基于对象的分布式操作系统，由 Andrew Tanenbaum 和他的学生们在荷兰 Vrije 大学开发。Amoeba 的微内核和它的进程管理在 [MVTVV90、TaAn90 和 Tan95] 中有描述。由于 Amoeba 是基于对象的，所以 Amoeba 中的进程是对象。进程管理中用来协助管理的一个关键特性是进程描述符。进程描述符有如下几个字段。

1. 一个字段定义准备就绪执行的 CPU 的体系结构。
2. 一个字段将退出状态传达给属主。
3. 依赖平台的线程描述符包括（至少）一个程序计数器和一个堆栈指针

有三个函数明显与进程管理有关。第一个函数是 `exec`，通过一个远程过程调用（见第 3 章）来创建一个新的进程，包括进程描述符。第二个函数是 `getload`，它通过搜索一个可用的处理器，并返回信息来帮助迁移。具体地说，它返回关于 CPU 速度、当前负载及可用的空闲内存空间的信息。最后一个函数是 `stun`，可在紧急和正常模式下操作，帮助挂起或关闭一个进程。收到一个正常的 `stun` 消息时，进程必须立即结束工作并回应。紧急 `stun` 消息则立即终止进程而不等待回应。如果被关闭的进程有子进程，它们将被挂起并成为孤儿进程。

2.3.1 进程类型

从调度者的角度来看，在分布式和并行环境中两大类进程。第一类是不可分割的进程，即所有进程是独立的、不可分割的，并且可能大小不同。对这些进程，不可能分割处理负荷，整个进程必须分配给单个处理器。这样的进程不能利用分布式和并行环境中额外的可用资源。

第二类是可分割的进程，可以根据算法特性分割成更小的子进程或任务（也就是线程）。

只有当所有任务都完成时，整个工作才完成。每个任务可作为一个独立的进程。某个进程的相关任务可以表示成一张图，其中顶点表示任务，边表示进程间的交互。这张图通常称为**任务交互图**（task interaction graph, TIG）。如果进程间有优先关系，图中的边可有方向（有箭头），也就是说一个进程必须优先于其他进程被执行。而且，这一组进程的关系还可能要求并发。详细说明 2.4 给出了一个可分割进程的例子，以及 TIG 的用法。

40

详细说明 2.4

可分割的进程及其 TIG

让我们来看一个普通的日常进程：制作和提供晚餐。它可分解成如下任务：

T1. 准备主菜

T2. 准备辅菜

T3. 准备餐后甜点

T4. 布置桌子。

T5. 上菜。

T6. 上甜点。

进程有优先关系，TIG 如图 2-5 所示

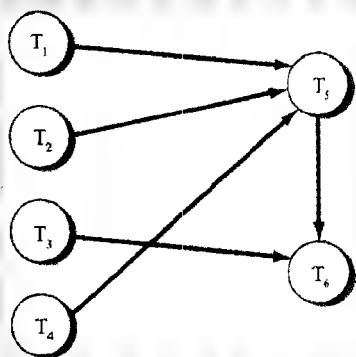


图 2-5 显示优先关系的 TIG

如描写和期待的那样，食物必须在端上桌子前先准备好。类似地，在上菜之前必须先布置桌子。最后，按习惯上先上菜后上甜点。

2.3.2 负荷分布和进程迁移

分布负荷的主要目的是更有效地利用资源和获得结果。这一目的也许应该称为**负荷平衡**，努力使系统的负荷达到相等的吞吐量，用来尽力帮助过载的资源，或者称为**负荷分担**。进程负荷的分布涉及到几个问题，包括利用远程处理器。负荷分布通过**进程迁移**来完成，将进程和所有相关的状态信息重新安置到另一个处理器上。如果进程是不可分割的，则整个进程必须迁移到另一个位置；如果进程是可分割的，则只有一部分必须被迁移。由于可分割的进程通常依靠改变共享数据，因此必须特别小心地判断可分割进程的一部分是否被分布到远程节点。当然，一旦一个进程迁移了，它就本地调度。

负荷分布的算法有两个部件，它们可以一起或不一起实现。所有部件可以用集中式算法或分布式算法来解决。第一个是信息收集部件，第二个是进程选择部件。信息收集部件负责决定收集什么信息，并且收集这些信息，然后选择一个适合的迁移对象。最成功的信息收集算法根据需求收集信息，或者当某个节点的状态改变了预定的数量（如 CPU 慢了 10%）或改变了负荷状态。根据实际的算法，系统可能有两个或三个负荷状态。两状态的算法认为一个节点要么过载，要么负荷不足。三状态的算法增加了一个合适加载的状态。

可用资源的辨认是一项复杂的任务。辨认算法也可以是集中的或分布的。集中式算法含有单个处理器（或节点），负责所有系统的所有信息，注意它们当前的负荷和可用资源。如果一个进程想迁移，它可以联系集中式管理者，索要整个系统的信息。这个集中式管理者必须高效操作和及时更新，因此最有效的方法是维护最少信息以使整个操作流畅。下面给出辨认算法必须回答的一系列问题。

- ◆ 所有进程平等对待，还是某些进程有更高优先级？
- ◆ 资源能否通过辨认一个 CPU 空闲了 x 微秒来辨认自己空闲或可用？
- ◆ 如何判断一个资源将空闲多长时间？在分布式系统，有闲置 CPU 的系统其用户是去喝杯咖啡，还是回家了？如果用户回来怎么办？
- ◆ 如果不搜索或发现空闲处理器，如何评估某个特定节点的负荷？这个评估是静态地还是动态地计算？
- ◆ 迁移能提高其他进程的本地性能吗？它对远程节点有消极影响吗？
- ◆ 远程资源的日常用户能杀死迁移进程吗？日常用户能强迫一个进程迁移到另一个节点吗？
- ◆ 进程迁移过程中发生系统故障的风险是什么？预防进程迁移过程中发生系统故障的代价是多大？
- ◆ 在负荷分布时，成员是自愿的还是强迫的？

分布式解决方案一般包含广播其需求或可用资源。因此一个节点可以发出一个消息，比如“我现在有一个空闲的 CPU，如果需要请通知我”。为了使系统免受广播信息的轰炸，一个节点可以广播“我需要另一个处理器，能帮我吗？”，然后那些有可用资源的节点就只跟那个需要帮助的节点联系。有了这个信息，就可以选择迁移对象。

现在我们确定了迁移对象，我们必须决定迁移哪个进程。因此，如前面所提及，第二个部件是进程选择部件。这个部件必须考虑几个问题，包括所要迁移给定进程的负荷量，给定进程对计算机体系结构的依赖性，给定进程的预期执行时间。如果负荷高而执行时间不长，则这个进程不是一个好的迁移选择。相反，如果进程几乎是新创建的（可能就是引起系统去搜索迁移对象的那个进程），它能被迁移并且在远程节点从头到尾运行。在远程节点完全执行比抢占后迁移的进程更能减少成本。下面是考虑进程迁移成本的部分要点列表。

- ◆ 迁移进程的初始通信延迟是多少？
- ◆ 返回进程结果到主节点的最终通信延迟是多少？
- ◆ 进程的各个部分如果没有迁移到同一个节点，各部分之间的通信成本是多少？
- ◆ 总的来回时间是否优于不迁移这个进程？
- ◆ 每个处理器的处理能力如何？
- ◆ 进程的各个部分如果没有迁移到同一个节点，如何维护各个部分之间共享的可变数据的一致性？

◆ 如何将虚拟内存随迁移进程一同转移（在第 4 章中讨论）？

◆ 如果系统中存在异构，如何协调其间的差异？

进程迁移涉及的另一个问题是在异构环境中操作的复杂性。异构包括使用不同类型的计算机，这就要求迁移必须能够处理不同速度的不同处理器、不同的软件配置，以及更困难的数据表示方法。不同的数据表达意味着迁移一个进程可能还包括数据翻译。Maguire 和 Smith [MaSm88] 提出了一个方法是使用一种外部数据表示（external data representation, EDR）。这个通用表示方法可被分布式系统中的所有参与成员使用，以减少翻译工作量。图 2-6 显示只有 5 种不同类型的计算机，却可能出现 20 种不同的翻译。如图 2-7 所示，通过将数据翻译成一个通用的外部数据表示，就只有 10 种翻译了。执行跨平台的进程迁移所需要的软件数量就可大大减少。翻译器要处理的最重要问题之一包括数字表示，因为不同的系统使用不同的尾数、指数和负数表示方法（零的表示方法）等。

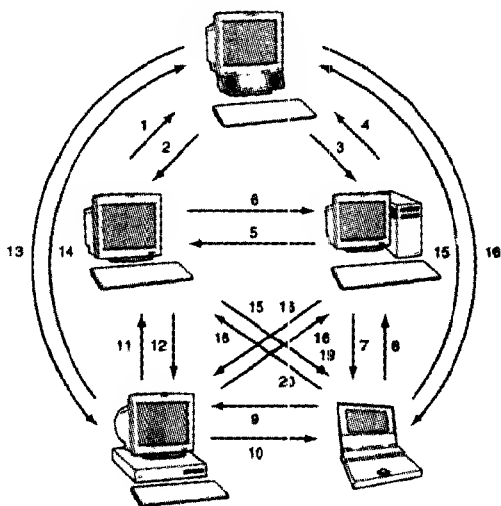


图 2-6 没有外部数据表示的数据翻译

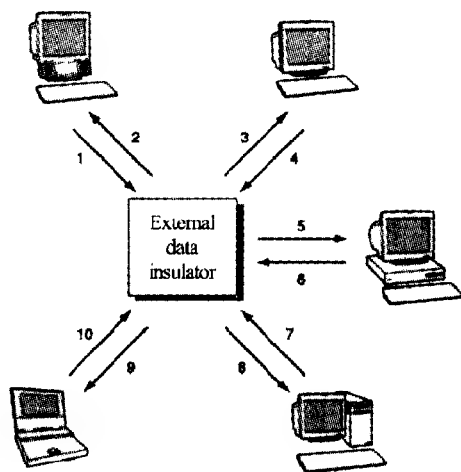


图 2-7 带有外部数据表示的数据翻译

还有一个问题是：何时该用集中式算法，何时该用分布式算法？在并行系统或分布式系统中使用一个处理器池和多个远程终端，为了对整个系统的情况更容易把握，因此宜于采用集中式算法，否则使用分布式算法更好。

2.4 进程调度

在带有多个处理器的环境中（并行或分布式），调度成为一件有意思的事。在实时环境中，时间不是无足轻重的，而是相当重要的。因此，分布式实时环境必须考虑通信延迟，并尽力符合它们的时间约束。在 2.4.1 节中我们来看看两种识别调度进程的方法。在 2.4.2 节解释调度器的各种组织方法。分布式调度算法更深入的讨论在第 7 章中，其它有关信息在第 9 章中讨论。

2.4.1 识别用于调度的进程

识别可调度的任务的最简单方法是所谓的轮询。在轮询设计中，调度器有一份完整的可

能需要系统服务的进程的列表。如名字所示，它在整体上设计成一个循环。这个循环不断询问每个进程是否需要系统。如果回答是，则符合条件；如果回答不是，则循环继续询问下一个进程。由于这个方法不是交互式的，这种设计更常用于特定目的的调度，而不是整个系统的调度。一个特定目的的调度可能控制访问一个网络或者其他 I/O 设备。详细说明 2.5 描述了一个使用轮询来调度访问网关的例子。

详细说明 2.5 轮询算法的例子

让我们来看看几台计算机连接到网关的简单例子。网关使用如下的轮询算法来识别哪个进程/机器将被同意访问网关，如图 2-8 所示。

```
current_machine = 0
Repeat
  If offer_service(machine[current_machine])
    // 如果系统需要访问
    Then
      service(machine[current_machine]); // 访问网关
    Endif;
    current_machine = (current_machine + 1)
                      mod total_number_of_machines;
Until (eternity)
```

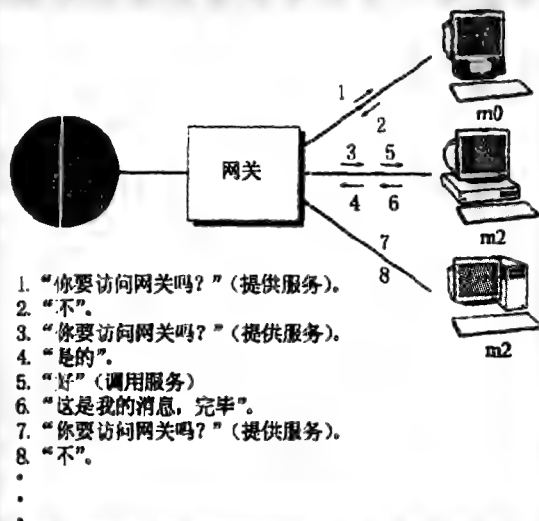


图 2-8 轮询的例子

对于集中式系统，高级系统可能是基于中断的。中断是发给系统的一个简单信号，表示需要某类设备。因此，中断也可用作识别调度进程的方法。不像轮询系统，中断并不保证可以立即访问想要的系统设备。相反，中断根据进程的优先级和请求的类型（根据中断的类型），将进程放入相应的调度队列。此外，还有第三个概念是时间中断，包括计时器的使用。每当一个特定中断出现，计时器就重置。如果一个中断没出现，而计时器又超时，则产生一个中断。

2.4.2 调度器的组织

有两种方法来组织调度器的整体功能。对于带有优先关系的可分割进程，调度器一般按状态驱动方式来组织。各种状态的组织依赖于进程及其各个部件之间特定的优先关系。如果未遵守优先关系，则最终结果可能不正确，并且不可避免地会污染数据。一组进程中存在优先关系，可能会要求全部并发运行。如果一个进程没有同时调度，整个工作可能就要花费两倍时间来执行。因此程序员必须通过调度算法来安排精确调度，否则调度器就是盲人瞎马。所有相关进程一起调度也常称为群调度 [Ous82]。详细说明 2.6 提供了表示精确调度优于盲调度的例子。

在一个分布式系统，如果负荷是可分割的，调度器必须注意所有相关成本。其中一个最重要的相关成本是由于通信而导致的反应时间延迟，包括把进程及其所有部分送到相应处理器所需要的时间。在分布式系统中，传输进程信息也会是一个重要的时间因素。在 NUMA 系统中如果进程被指派到其他地方执行的话，调度器还必须考虑获取和移动数据所需的通信成本。

详细说明 2.6 并发进程调度

阶段 进程	0	1	2	3	4
1	W	R	R	R	E
2	W	R	R	R	E
3	W	R	R	R	E
4	W	R	R	R	E
5	W	R	R	R	E

解释：R = 运行
W = 就绪
B = 阻塞
E = 退出

进程 1~5 是相关的

图 2-9 精确并发调度

阶段 进程	0	1	2	3	4	4	4
1	W	R	B	B	R	R	E
2	W	R	B	B	R	R	E
3	W	R	B	B	R	R	E
4	W	W	R	B	R	R	E
5	W	W	W	R	R	R	E

解释：R = 运行
W = 就绪
B = 阻塞
E = 退出

进程 1~5 是相关的

图 2-10 盲并发调度

上面的例子描述了对相关进程进行适当调度的好处。在精确调度中，关于进程的信息从访问调度例程的程序员处获得。在盲调度中，程序员没有提供任何信息给调度器。

假设我们有五个相关进程。这些进程都产生可被下一阶段相同进程利用的信息。执行一个阶段所需的信息依赖于前一阶段完成的所有进程。这种关系称为生产者/消费者关系。我们假定这五个相关进程有三个基本执行阶段。图 2-9 显示了精确调度的结果，而图 2-10 显示了盲调度的结果。在图 2-10 中可看出，进程 1 到 3 被迫阻塞而不能进入第二和第三阶段的执行，直到进程 4 和 5 也被调度。

状态驱动调度器常常通过协同例程来实现，具体地说，进程所有处于特定状态的方面都被调度。当一个状态完成后，调度器就为下一状态调度适当进程。状态驱动调度器的一个例子是 IBM 的 OS/2 Presentation Manager™。这个程序使用协同例程来调度进程，并维护协同及调度用户窗口所需要的优先关系。优先关系方面的知识将在第 9 章讲述。

如果没有优先关系，调度器的组织就用我们所学过的集中式操作系统同样的方法。具体地说，它可以使用诸如循环调度、最短工作优先调度、优先权调度（包括派生的前景/背景和剥夺优先权调度）这样的算法。

2.5 小结

内核是操作系统中非常重要的部分。在本章，我们已经介绍了一些基本的概念，这些概念中的大部分将在第 3 章和第 4 章中进一步深入学习。并行和分布式系统倾向于做成微内核，这种设计特别适用于需要可移植性和灵活性的分布式系统。要求效率的实时系统也使用微内核。内核负责许多事情。在本章我们着重分析了进程和线程的概念，以及进程管理和调度。进程是可以分割的，由多个任务组成，并拥有几个执行线程。在分布式系统，内核不仅决定进程将在哪里执行，还要负责本地调度。为了与内核及其他进程通信，需要用到进程间通信，这方面将在第 3 章详述。内核也负责内存管理，这方面将在第 4 章讨论。

2.6 参考文献

下面的参考文献可提供关于内核及其功能和实现的更广泛信息。[Al197, BGMR96, Cha97, Cou98, Fla96, Kri39, KSS97, Lap97, LeBe96 和 RoRo96]。与本章中内容有关的一些传统研究论文有 [ABLL91, ArFi89, BrJo94, Caku88, Cve87, EAL95, ELZ86, Esk89, Fer89, FeZh87, GeYa93, KrLi87, Kun91, LHWS96, Lo88, Mul93b, PTS88, SKS92, SWP90, TaKa94, VLL90, WaMo85 和 Zay87]。

下面提供一些与内核问题有关的 Internet 站点资源。Anderson、Bershad、Lazowski 和 Levy 的论文 [ABLL91] 可在 <http://www.cs.berkeley.edu/~brewer/cs262/Scheduler.pdf> 找到。Sun Microsystems™ 提供了一个专注于线程的主页 <http://www.sun.com/software/Products/Developer-products/sig/threads/index.html>。Sun 的站点也包含使用线程的程序范例，可在 <http://www.sun.com/workshop/sig/threads/Berg-Lewis/examples.html> 看到。另一个关于线程的主页地址为 <http://www.mit.edu:8001/people/proven/pthreads.html>。Amoeba 主页地址为 <http://www.am.cs.vu.nl>。

习题

- 2.1 指出并说明相对于单内核，使用微内核设计高级系统内核的三个主要优点。
- 2.2 对下面每个线程结构范例列出一个可能的应用。
 - a. 专家范例。
 - b. C/S 范例。
 - c. 流水线范例。
- 2.3 指出并说明内核支持的线程的优点。
- 2.4 指出并说明用户级支持线程的优点。
- 2.5 何时不利于切换出一个阻塞进程？
- 2.6 假设有一个进程要迁移，进程已经在 UNIX 环境下编译执行。请说明将这个新进程迁移到如下各个节点的好处和随之产生的问题。
 - 节点 1
 - ◆ UNIX 环境。
 - ◆ 每个方向的通信要花费 150 个时间单元。
 - ◆ 处理器当前使用 80%。
 - ◆ 处理器速度为 100MHz。
 - 节点 2
 - ◆ UNIX 环境。
 - ◆ 每个方向的通信要花费 250 个时间单元。
 - ◆ 处理器当前使用 60%。
 - ◆ 处理器速度为 60MHz。
 - 节点 3
 - ◆ Windows98™ 环境。
 - ◆ 每个方向的通信要花费 50 个时间单元。
 - ◆ 处理器当前使用 30%。
 - ◆ 处理器速度为 200MHz。
 - 节点 4
 - ◆ Windows-NT 环境。
 - ◆ 每个方向的通信要花费 1000 个时间单元。
 - ◆ 处理器当前使用 20%。
 - ◆ 处理器速度为 260MHz。
 - 节点 5
 - ◆ Windows3.1™ 环境。
 - ◆ 每个方向的通信要花费 10 个时间单元。
 - ◆ 处理器当前使用 0%。
 - ◆ 处理器速度为 60MHz。
- 2.7 编写一个程序来进行迁移决策。这个程序应该适应 OS 环境的类型、通信成本、处理器应用和处理器速度。此外，用户应该能够为这些因素的重要性分别加权。

根据用户要求的重要性加权（所有因素的加权数之和是1.00），判断练习2.6中哪个站点是最佳的迁移对象。程序应该使用如下公式：

$$\text{Total_Weight} = [\text{Env_WT} * (1/0)] + [1 - (\text{Com_WT} * \text{Com_Cost})] + [1 - (\text{Util_Wt} * \text{Util})] + [\text{Speed_Wt} * \text{Speed}]$$

对于环境加权（1/0），如果环境匹配，则它乘以1；如果环境不匹配，则它乘以0。

用练习2.6中的五个节点和下面的权值来测试程序。

权值1

- ◆ 环境 .25。
- ◆ 通信成本 .25。
- ◆ 处理器当前使用 .25（越低越好）。
- ◆ 处理器速度 .25（越高越好）。

50

权值2

- ◆ 环境 .75。
- ◆ 通信成本 .25。
- ◆ 处理器当前使用 0（越低越好）。
- ◆ 处理器速度 0（越高越好）。

权值3

- ◆ 环境 .55。
- ◆ 通信成本 .15。
- ◆ 处理器当前使用 .20（越低越好）。
- ◆ 处理器速度 .10（越高越好）。

权值4

- ◆ 环境 .15。
- ◆ 通信成本 .55。
- ◆ 处理器当前使用 .20（越低越好）。
- ◆ 处理器速度 .10（越高越好）。

2.8 假设迁移策略允许资源的属主立即（不经警告）杀死所有在它的资源上运行的进程（比如用户稍息片刻后返回了），会影响下面哪些项目？

1. 进程的执行进展。
2. 执行进程的状态信息。
3. 执行进程驻留在系统内存中的数据。
4. 整个分布式系统的整体效率。

2.9 在一个使用7种不同体系结构，而又不采用EDR的异构环境中，需要多少种翻译？选做题：有和没有EDR所需翻译数的公式是什么？

2.10 假设系统为线程调度提供一个二级优先权系统：低优先权给用户级线程，高优先权给内核线程。调度器使用带有优先抢占的最短工作优先调度算法（任何新的高优先权工作到达都会抢占正在运行的低优先权工作）。计算每个线程总的加权时间和每类线程的平均等待时间。所有标记U的线程是用户级线程，标记K的是内核

51

级线程。优先权相同的线程按先来先用的原则调度。

到 达 时 间	线 程 编 号	需要的处理时间
0	U1	12
3	K2	5
9	K3	3
11	U4	5
27	K4	7
27	U5	2

2.11 写一个程序实现练习 2.10 所描述的带有优先抢占的最短工作优先调度，并用练习 2.10 中所给出的数据和你自己的数据来测试这个程序。

52

2.12 说明系统调度使用轮询调度器的两个缺点。

第3章 进程间通信

要顺利地将各部分组织在一起，一个重要的因素是建立有效的通信方式。因此，如果我们要在任何高级环境中获得用多个进程处理同一个问题的好处，就一定要使用适当和有效的通信方式，尤其是**进程间通信**（interprocess communication, IPC）。这是一个基本的概念，甚至用于在最简单的计算机游戏中，以允许用户通过网络参与。在本章中，我们讨论各种类型的进程间通信方法。这些方法在底层实现并涉及系统调用，或通过语言支持实现。我们讨论进程间通信的四个主要类型。3.1节介绍了在分布式系统中实现进程间通信时能影响机制选择的几个因素，3.2节介绍消息传输，3.3节研究管道，3.4节讨论套接字的细节，3.5节介绍远程过程调用。第4章讨论另一种通过使用共享内存来进行通信的方法。CORBA和DCOM提供的对象间通信方式将在第6章讲述。

53

3.1 选择因素

影响程序员为某一给定的应用选择进程间通信方法的因素有很多，以下是一些应该考虑的问题：

- ◆ 程序员当前的技能（这虽不是选择某一特定方法的具体原因，但它确实影响许多系统）。因为如果你以前实现过某种IPC方法，那么再次去实现就简单多了。
 - ◆ IPC机制对程序员的透明性。程序员知道的细节越少，出错的几率也就越低，但同时也增加了从细节中抽象和自动处理它们的费用。
 - ◆ 组成分布式环境的各个系统所支持的方法。很显然，除非该机构客户定制了结构框架，否则不容易选中不被需求环境所支持的方法。
 - ◆ 系统将来的可扩充性。这个应用是否将一直运行于C/S模式下？我们是否会在超越单机和/或文件空间的范围通信？如果你的选择限制了超越当前的文件空间的能力，就要牺牲将来的扩展性。（记住：许多原型直接应用到产品中，因此最好不要假设还会有重写原代码的情况）。
 - ◆ 支持进程的移植。如果一个IPC方法禁止不同文件系统间的通信，那么进程就不能在文件系统域之外进行移植。
 - ◆ 机制是否已经标准化，从而最大程度地允许各种系统能够加入分布式系统中，并都具有执行通信的能力？
 - ◆ 机制的效率。这在具有时间限制的分布式实时系统中尤其需要。
- 记住以上几点，我们开始讨论各种IPC方法。

3.2 消息传递

消息传递（message passing）允许两个进程间通过将共享数据物理地拷贝到另一进程的地址空间来进行通信，这通过向包含共享数据的进程传送消息来完成。这种形式的通信在不共享内存空间的两个进程间非常普遍，只要它们具有各自的内存，而不管在物理上是不同的

54

系统还是相同的系统。消息传输通信包括类似如下形式的发送和接收原语。

```
send (b, msg);
receive (a, msg);
```

这些可以是阻塞或非阻塞原语。为了理解阻塞和非阻塞原语之间的区别，让我们先看一个假设的情景：苏珊打电话给史蒂芬并留下信息，信息中包含一个问题并要求回电。这种情况下，苏珊有两种选择，坐在电话旁边等待史蒂芬的回电或者是做其他工作直到接到上述的回电。第一种情况类似于阻塞原语，第二种情况则类似于非阻塞原语的消息传输情况。每种选择都有其优缺点，我们将分别在 3.2.1 和 3.2.2 节中进行描述。另外，消息可能传输至一个或一组进程，这种情况将在 3.2.3 节中进行描述。

3.2.1 阻塞原语

假设苏珊留给史蒂芬的信息是他们准备一个令莉恩惊喜的聚会，苏珊和史蒂芬决定分工如下：

苏珊的任务：

1. 预订蛋糕。
2. 取蛋糕。
3. 让史蒂芬知道蛋糕已经取回。
4. 接史蒂芬的电话，得知莉恩因宴会而惊喜。
5. 在莉恩惊喜后把蛋糕带到聚会地点。

史蒂芬的任务：

1. 打电话给莉恩。
2. (确认蛋糕已经取回。)
3. 把莉恩带到宴会上。
4. 让苏珊知道莉恩在聚会地点，并且达到目的。

如果苏珊没有等史蒂芬的回电，那么她也许会在进入宴会厅之前就在入口处碰到他们而破坏了惊喜的效果。因此，苏珊和史蒂芬不但需要通信，而且还需要同步。

如同苏珊和史蒂芬，当多个进程解决同一任务时，它们不仅需要通信，还需要同步。因此对于这些进程来说，一个阻塞消息传输的原语，也称同步原语，可以完成要求。阻塞原语不立即返回控制权给进程。一个 send 阻塞直到确认为止，而 receive 阻塞直到消息被接收。当一个进程被阻塞，它在解除阻塞之前不能继续执行。因此，两个进程不仅需要通信而且还需要同步。如果你想执行同步，可能它很有用，但是阻止了并发性。

阻塞原语可以使用缓冲，也可以不使用缓冲。当使用时，缓冲区也称为邮箱。在各自的进程阻塞解除之前，对发送的确认或接收到的消息放置在一个缓冲中。这个缓冲可位于内核的地址空间也可在进程的地址空间。在分布式环境中，缓冲由接收者的位置确定，这也产生了一个问题：如果接收者还没有发送 receive 命令，内核就不知道该把消息放在何处！

当阻塞原语执行时没有缓冲，那么在内核发出 receive 通知之前，传输被阻塞。一旦收到这个通知，它就会继续执行传输。

一般有两种方法可以让进程知道阻塞已解除。一种方法涉及轮询。当一个进程使用轮询时，它执行一个测试原语来根据缓冲的状态决定缓冲中是否有相关信息。这需要不停地检查

缓冲, 因而占用 CPU 的资源, 这种行为称为忙等待。然而许多应用都不受忙等待的影响, 它们并不是实时应用所需要的。另外, 轮询一般不在间隔小于 1 分钟的系统中使用, 否则会影响系统的性能。另一种方法涉及中断的使用。中断也可用于非阻塞原语, 因为进程不需要一直浪费时间去检查缓冲, 所以使用中断效率更高。

由于消息传输要考虑同步, 所以传送和接收原语必须阻塞。如果一直测试失败或没有中断, 那么进程会一直阻塞。因此不管使用哪种方法, 必须使用计时器, 计时器可以使用默认值或由程序员控制。在没有缓冲的实现中, 每次执行发送原语时计时器开始计时, 如果在计时结束时还没有得到内核的通知, 则进程重试有限次, 如图 3-1 所示。在分布式环境中, 通过网络不断地重传消息是不现实的, 因此在分布式环境中一般都至少有一个单一缓冲。详细说明 3.1 描述了图 3-2 中使用缓冲的阻塞发送和阻塞接收。

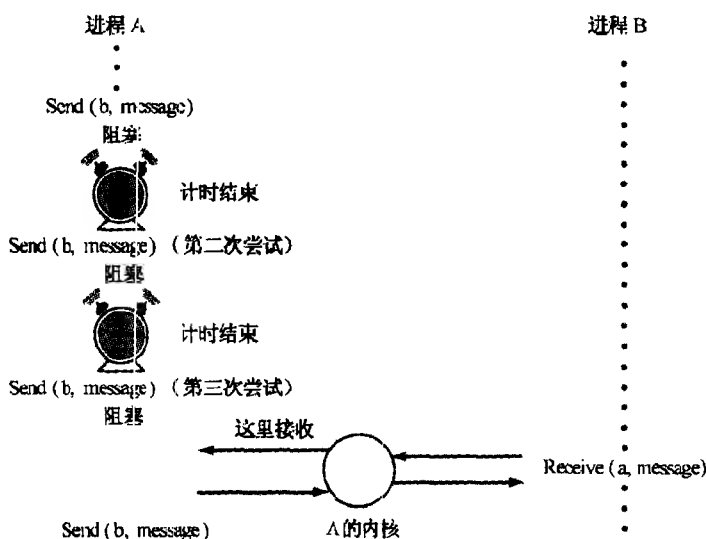


图 3-1 阻塞发送和接收原语：无缓冲

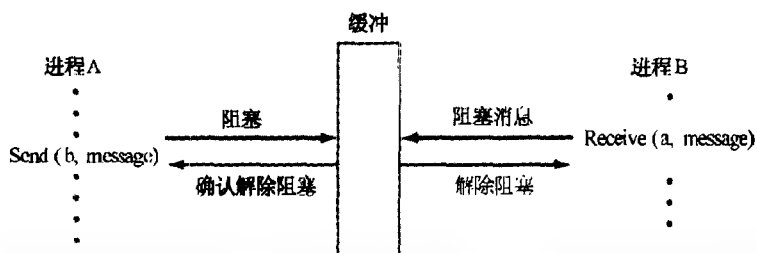


图 3-2 带缓冲的阻塞发送和接收原语

详细说明 3.1

使用缓冲的阻塞发送和阻塞接收

通常阻塞发送相当于如下所示：

Procedure A
Begin

// 过程 A 开始

```

Instructions
...
send (b, message,...)      // b是目标地点
                           // 等待确认
                           // 接收发送确认
next instruction
...
End                        // 过程 A 结束

```

通常阻塞接收相当于如下所示：

```

Procedure B
Begin                      // 过程 B 开始
Instructions
...
receive (a, message,...)   // a 是源地
                           // 等待消息
                           // 接收消息
next instruction
...
End                        // 过程 B 结束

```

3.2.2 非阻塞原语

大多数人都会认为停止其他一切生产活动而坐在电话旁等电话的效率很低，进程也不希望阻塞和阻止并发，因此同步有时不必要也不需要。当非阻塞原语执行发送和接收时，控制权立即返回给进程并继续执行语句。然后，收到响应而产生了一个中断，通知进程可以完成原语。因为这种方法不提供任何形式的同步，所以非阻塞原语也称为异步原语。

类似阻塞原语，发送进程通常可以通过消息或中断重用缓冲，这有助于缓解发送者在发送之前就重写消息而带来的问题。由于进程使用非阻塞原语时，原来使用阻塞原语的进程在阻塞的时候可以从事其他工作或执行语句，所以它在实时应用中非常受欢迎。

3.2.3 进程地址

消息传输系统中的编址分为两类：一对一通信编址和组编址。

一对一编址

一对一编址可以是显式的，也可以是隐式的。**显式编址**需要在通信的进程中显式地命名参数，如图 3-3 所示。**隐式编址**则只需把服务名称作为参数包含在发送消息中而不是具体的进程，如图 3-4 所示。因此，任何（只有一个）C/S 关系中的服务器都会对通信消息作出响应。隐式编址的 send 原语也称为 send_any 命令。类似地，在 receive 原语中的隐式编址说明服务器准备与任何客户机通信，隐式编址的 receive 原语也称为 receive_any 命令。

组编址

组编址是为支持组通信而设计的，在并行和分布式系统中很有用。组通信可以分为三类：

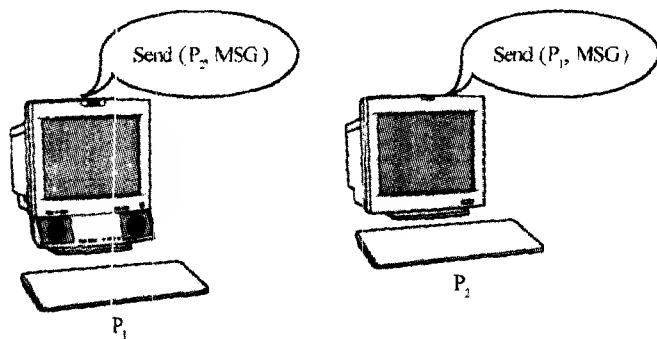


图 3-3 进程间通信的显式编址

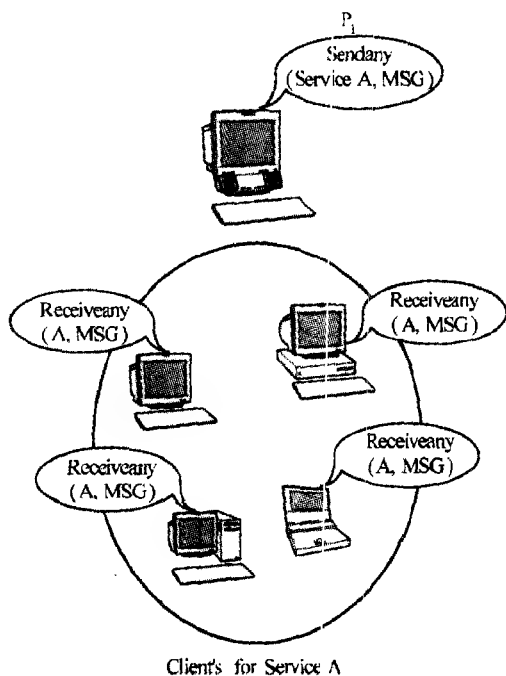


图 3-4 进程间通信的隐式编址

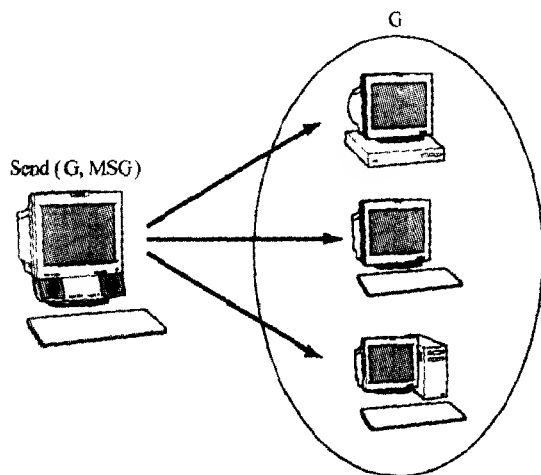


图 3-5 一对多组编址

- ◆ 一对多。在这种情况下，有多个接收者，但只有一个发送者。因而使用这种编址类型的通信也称为组播通信。如果所有的接收者都在网络上，并且网络上所有的成员都是接收者，那么它也可以称为广播通信。与一对一的隐式编址不同，所有接收组的成员都收到消息。这种类型的编址有很多应用，包括定位一个可用的服务器或通知某一接收组一个问题。图 3-5 是一对多的组编址。
- ◆ 多对一。在这种情况下，有多个发送者，但只有一个接收者。与一对一隐式编址相类似，惟一的接收者可以从一组中的一个发送者那里接收消息，也可以从一组的几个或所有的成员那里接收消息。
- ◆ 多对多。在这种情况下，有多个接收者，也有多个发送者。这种类型编址的最大困难

是进行消息排序，见详细说明 3.2。

详细说明 3.2 多对多消息排序

在使用多对多编址通信时，许多应用都要求消息以一种可接受的方式接收。我们现在从结构性最差的到最严格的语义顺序来讨论三种基本的排序语义：

1. 随机排序。结构最差但最快的排序语义。它保证了随机相关的消息按正确的顺序被接收。该方法与 Lamport 的算法相关，后者将在第 10 章中讨论。

2. 统一排序。它要求所有的消息被所有的接收进程统一接收，即以同样的顺序被接收。该顺序可以与发送的顺序不同，如图 3-6 所示。

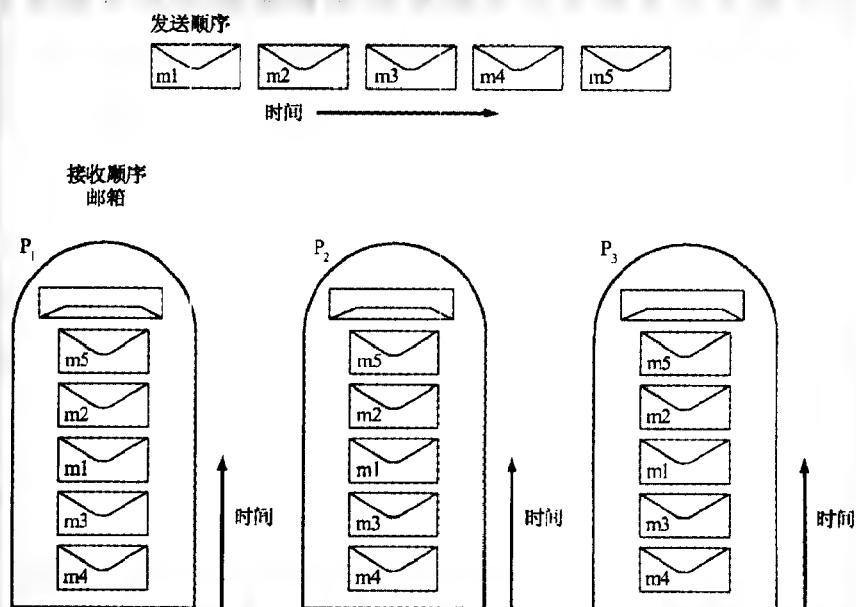


图 3-6 统一排序

3. 通用排序。最严格的语义，它要求并且保证所有的消息严格按照发送的顺序被接收，因而需要一个通用时间，也就是说进程所涉及的时钟必需同步（如果它们不在同一系统中），或者使用全局时间戳，详见第 10 章。

3.3 管道

管道是最早的、最原始的进程间通信机制。它允许两个进程通过由内核提供的有限缓冲进行通信，如图 3-7 所示。通信用的数据按先进先出（FIFO）次序进行存储。这种通信原语由系统调用实现。管道提供一对一通信，并且只在通信的过程中存在。传统的 UNIXTM管道是单向的，但现在的版本中是全双工或双向的。当缓存写满时引起的进程阻塞的管道认为是同步的。它有两种类型：非命名管道和命名管道。

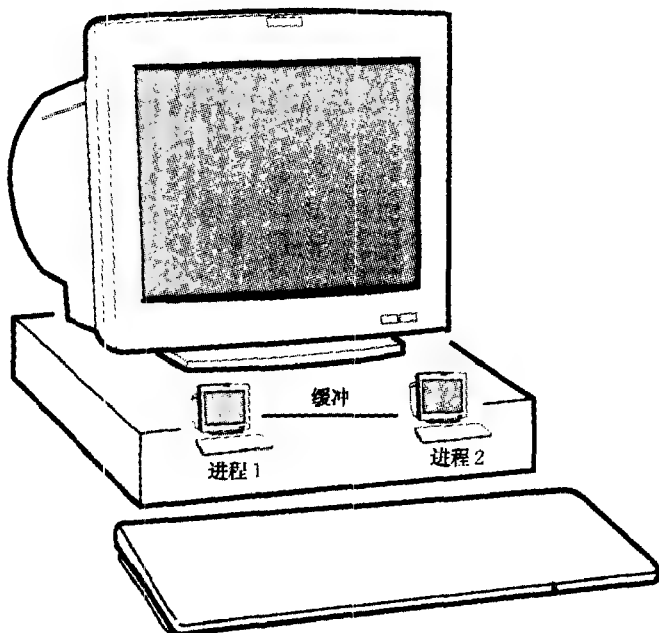


图 3-7 使用管道的进程间通信

3.3.1 非命名管道

非命名管道允许两个相关的进程，比如由 `fork` 操作产生的进程进行相互通信。相关的进程包括父/子和兄弟进程。在 UNIX 中，这种管道可以使用系统调用 `pipe` 或在命令行使用竖杠 (`|`) 如 `ls|more` 来产生。管道在 Windows 环境中也很普遍。

3.3.2 命名管道

命名管道允许不相关的进程进行通信。即使进程间不需要相关，这些进程也需要共享共同的文件系统。因为命名管道使用共享的数据结构，所以必须在临界区内访问缓冲，详见第 5 章。详细说明 3.3 讲解了如何在 UNIX 环境中使用命名管道。

详细说明 3.3 UNIX 命名管道

在 UNIX 中，命名管道的建立产生了目录的入口，在这个入口上文件访问权限允许相关进程通过这个命名缓冲进行通信。与非命名 UNIX 管道类似，命名管道能够在命令行中通过命令 `mknod` 建立，例如：

```
% mknod MYPIPE p
```

其中 `mknod` 是命令，第一个参数是命名管道的名称 (`MYPIPE`)，第二个参数 (`p`) 说明它是一个命名管道。一般管道名称全部使用大写提醒用户这是一个特殊文件。创建好以后，

用户可以使用 UNIX 的命令或 `chmod` 改变访问权限。众所周知，UNIX 系统中管道是 FIFO 的，`mknod` 命令经常要求超级用户权限，但 UNIX 系统中包括一个叫 `mkfifo` 的 C 库函数来辅助产生命名管道并不要求这种权限，`mkfifo` 的函数头如下：

```
int mkfifo (const char * path, mode_t mode);
//POSIX.1 Spec 1170
```

`path` 是创建的命名管道的路径名，`mode` 规定用户、组和其他访问所创建的管道的权限。如果创建管道成功，返回值是 0，否则为 -1。

以下代码段允许程序创建一个对所有用户只读的命名管道。

```
#include <sys/stat.h>
#include <sys/types.h>
mode_t fifo_perms =
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
if (mkfifo("MYPIPE", fifo_perms) == -1)
    perror("Couldn't create MYPIPE");
// 建立了名称为 MYPIPE 的命名管道
```

3.4 套接字

为达到不同域的进程间通信，或者不共享数据结构或文件的系统通过网络的通信，就必须使用传输服务的一种机制，最广泛使用的机制是套接字。所有的套接字都由传输服务管理，每个套接字有两个通信的端点，每个端点属于通信进程中的一个。每个套接字有一个局部的端点地址和一个全局的端点地址。局部地址指传输服务中的地址，而全局地址是指网络主机地址，这两个地址通过系统调用绑定在一起。这两个地址都是必需的，除非在通信之前已放置一个连接套接字调用。这个调用可以把局部地址与远程全局地址绑定，从而简化了将来的通信。两个端点都必须执行绑定，一个套接字在使用它的每个进程死亡或一个进程关闭它之前都是存在的。在安全套接字层（SSL）上的信息提供了安全套接字的实现，详见第 11 章。

使用一个套接字一般需要 6 步，如图 3-8 所示。这一点在附录中使用套接字的外科手术调度程序中也有所体现。现在以电话作为类比来讨论一下每一步的情况。第一步，创建一个套接字。这类似于安装一个电话插座，就像在使用电话前必须有电话插座一样，在使用套接字进行通信前必须创建一个套接字。第二步，把套接字绑定在端口上。这类似于从电话公司获得一个与电话插座相关联的电话号码，只有把套接字绑定在一个地址上或把电话号码绑定在电话插座上才可以进行通信。第三步，侦听电话铃声或侦听套接字等待他人来进行通信。一旦意识到有人试图进行联系，就可以接受通信。第四步，如电话具有呼叫等待特征一样，套接字可以决定接受多少个呼叫。第五步是通信。这时可以开始进行通信，对套接字来说是通过读写命令来实现。第六步是在通信完成后断开连接或挂上电话。

套接字有两种基本的域类型。一种是流套接字（在 TCP/IP 中实现），用于高可靠性的面向连接的通信。另一种是数据报（在 UDP/IP 中实现），用于不可靠的非连接通信。由于数据报不需要额外负担来保证可靠性，所以这种类型更有效，对实时应用很有吸引力。Windows

第一步：创建套接字

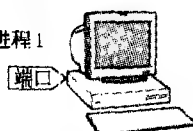
我需要一套接字

进程 1



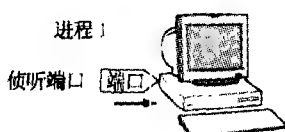
第二步：绑定

进程 1



第三步：侦听

进程 1

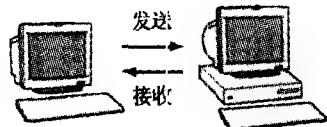


第四步：接受

进程 1

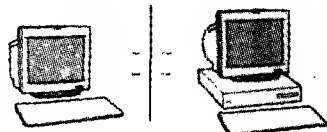


第五步：通信

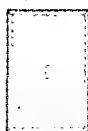


第六步：断开连接

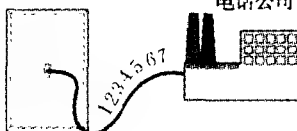
断开连接



类比



安装电话插座



侦听



回答电话



挂断

图 3-8 套接字的类比

环境中套接字的实现使用了 Winsock，它基于 UNIX 的 BSD 实现方法，详见 3.4.1 节。3.4.2 节详细讨论了使用 Java 实现套接字。

3.4.1 UNIX 套接字

最早的 UNIX 套接字接口起源于 1981 年的 BSD UNIX 3.1。用户必须使用 `-lsocket` 选项

来编译程序和使用 UNIX 套接字库。在 C 库中有 UNIX 的套接字原语，包括 socket, bind, connect, listen, send, receive 和 shutdown。UNIX 中允许两种类型的套接字，UNIX 套接字和 Internet 套接字。接口如下：

```
SOCKET
#include <sys/types.h>
#include <sys/socket.h>
int socket (int domain, int type, int protocol);
```

域参数在 UNIX 域中的值为 AF_UNIX，在 Internet 域中的值为 AF_INET。UNIX 域说明双方进程必须在单个 UNIX 系统中。类型参数在使用流的面向连接的可靠通信中取值 SOCK_STREAM，在使用数据报的非连接不可靠通信中取值 SOCK_DGRAM。流一般在 TCP 中实现，而数据报则在 UDP 中。通常，对于某一特定类型只有一种协议，因而一般协议值为其默认值 0。

```
BIND
#include <sys/types.h>
#include <sys/socket.h>
int bind(int s, const struct sockaddr *address, size_t address_len);
```

参数 s 是调用套接字函数的返回值，是一个文件描述符；address_len 是 * address structure 的字节数，包括地址类别和协议信息。当使用 Internet 域时，用于 struct sockaddr 的 sockaddr 的定义如下：

```
struct sockaddr_in
{
    short      sin_family;
    u_short    sin_port;
    struct      in_addr sin_addr;
    char       sin_zero[8];
};
```

在 Internet 域中，sin_family 参数的值与套接字函数中的值一样，是 AF_INET，端口值存储于变量 sin_port 中。如果允许同任何主机通信，sin_addr 应设置为 INADDR_ANY。为了使结构的大小与 sockaddr 相同，就用 sin_zero 填充。

当使用 UNIX 域时，sockaddr 使用如下的定义：

```
Struct sockaddr
{
    short sun_family;
    char  sun_path[];
};
```

在 UNIX 域中，sun_family 参数应像在套接字函数中一样定义为 AF_UNIX，UNIX 的路径名赋给字段 sun_path。

```
LISTEN
#include <sys/types.h>
#include <sys/socket.h>
int listen(int s, int backlog_size);
```

此函数只用于建立基于连接的套接字。同样，s 是调用套接字函数的返回值。字段 backlog_size 允许用户限制一个连接被拒绝之前所能等待的客户端连接请求数，这样可以允许一个服务器处理多个通信请求。

```
ACCEPT
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s, struct sockaddr *address, int *address_len);
```

我们注意到这些参数和 bind 函数非常相似，除了 *address 包含进行连接的客户端信息。所以在 Internet 域中，accept 中的 *address_len 指明了呼叫前的缓冲区的大小以及呼叫后填充到缓冲区中的信息的大小。返回值是一个用于和客户端通信的文件描述符。

```
GETHOSTBYADDR
#include <sys/types.h>
#include <sys/socket.h>
struct hostent *gethostbyaddr(const void *addr, size_t len, int
type);
```

该函数用于将 accept 函数中收到的地址转换成名字，因此非常有用。

```
SEND
#include <sys/types.h>
#include <sys/socket.h>
int send(int s, const char* buf, int len, int flag);
```

buf 中长度为 len 个字节的消息发送到一个由 s 指定的套接字中。

```
RECV
#include <sys/types.h>
#include <sys/socket.h>
int recv(int s, char* buf, int len, int flag);
```

同样，一条信息由套接字 s 接收并拷贝到 buf 中。接收信息的最大长度由 len 指定。所有的消息是以 FIFO（先进先出）的次序接收。如果设置了标志，那么表明接收不按照此次序（被分行）的带宽外（out-of-band）数据。这种消息应只用于紧急情况。

68

```
CLOSE
#include <sys/types.h>
#include <sys/socket.h>
int close(int s);
```

该函数用于关闭一个套接字 s。

最后，由于所有系统均有细微的差异。我们可能必须在发送之前转换数据的格式，尤其是某些机器使用 ASCII 格式而另外一些机器使用 EBCDIC 格式。这些机器也使用不同的字节顺序。下面是一些可用的转换函数。

htons ()：主机字节顺序转成短整型的网络字节顺序。
 ntohs ()：网络字节顺序转成短整型的主机字节顺序。
 htonl ()：主机字节顺序转成整型的网络字节顺序。
 ntohl ()：网络字节顺序转成整型的主机字节顺序。
 htonl ()：主机字节顺序转成长整型的网络字节顺序。

ntohl ()：网络字节顺序转成长整型的主机字节顺序。

3.4.2 Java 对套接字的支持

Java 提供了两个类支持 Internet 套接字。java.net.Socket 是客户端使用的类，而 java.net.ServerSocket 是客户端使用的类，这是使用套接字编程最简单的方法。客户端的类见详细说明 3.4。

详细说明 3.4

客户端 Java 套接字类

```

Public final class Socket extends Object{
//          公共构造函数
//
    public Socket(String host, int port)
        throws UnknownHostException, IOException;
//          使用主机名和默认的数据流
    public Socket(String host, int port,
        boolean stream)throws IOException
//          使用主机名和非默认的数据报
    public Socket(InetAddress address, int port)
        throws IOException;
//          使用 IP 地址和默认的数据流
    public Socket(InetAddress address, int port,
        boolean stream)throws IOException;
//          使用 IP 地址和非默认的数据报
//
// 类的方法
//
    public static synchronized void
        setSocketImplFactory(SocketImplFactory
        fac)throws IOException, SocketException;
//
// 公共实例方法
//
    public synchronized void close()
        throws IOException;
    public InetAddress getInetAddress();
    public InputStream getInputStream()
        throws IOException;
//          读套接字
    public int getLocalPort();
    public OutputStream getOutputStream()
        throws IOException;
//          写套接字
    public int getPort();
    public String toString();
//          重写 object.toString()
}

```

使用这个类，客户端首先通过构建一个套接字的方法与服务器建立一个连接，这种情况下，用户必须指定服务器的主机名和端口号。这个类的默认方式是使用流套接字，当然也可以指定使用数据报。

以下代码段试图与主机 `rose.myuniversity.edu` 的端口 25 连接一个套接字。当然如注解所示，可以根据情况把主机名替换成真实的 IP 地址，这样可以与没有静态 IP 地址的伙伴通信。

```
Try {Socket sock = new Socket
    ("rose.myuniversity.edu",25);
}
catch (UnknownHostException e)
{
    System.out.println("Unable to locate host.");
}
catch (IOException e)
{
    System.out.println("Host Connection Error.");
}
```

如果连接成功完成，用户可以通过如下语句：

```
Socket getInputStream () & Socket getOutputStream ()
```

检索输入和输出流来发送和接收信息。

当客户端建立连接之后，服务器端可以使用详细说明 3.5 中定义的 Java 服务器套接字类与客户端进行通信。

69
70

详细说明 3.5

服务器端 Java 套接字类

```
Public final class ServerSocket extends object{
//
// 公共构造函数
//
    public ServerSocket(int port)
        throws IOException;
    public ServerSocket(int port, int count)
        throws IOException;
//
// 类的方法
//
    public static synchronized void
        setSocketFactory (SocketImplFactory fac)
            throws IOException, SocketException;
// 公共实例方法
    public Socket accept() throws IOException;
//
    public void close() throws IOException;
```

```

//                                     关闭套接字
public InetAddress getInetAddress();
public int getLocalPort();
public String toString();

//                                     重写 object.toString()
}

```

71 ServerSocket 使用方法 accept 在一定的时间范围内连接到指定的端口建立侦听，端口号应大于 1024，因为标准的、众所周知的，以及系统的进程使用较小数目的端口号。然后服务器使用 getInputStream 和 getOutputStream 发送和接收数据，一旦创建了 ServerSocket，服务器套接字可以接受很多连接。

3.5 远程过程调用

许多计算机科学家习惯于使用过程的方式进行编程。远程过程调用（Remote procedure call, RPC）允许进程使用简单的（熟悉的）过程调用，通过网络与远程进程通信。从前面介绍的进程间通信技术可以看出，在通信进程之间只有简单的消息传输。像许多流行语言的过程一样，RPC 不仅允许程序员通过发送消息进行通信，还允许更复杂的数据结构作为参数值和返回值。调用者在等待返回值时被阻塞，所以 RPC 以并行为代价提供同步，如同阻塞的消息传递原语一样。因为 RPC 的调用函数等待的不仅仅是一个确认，还包括返回值，所以调用函数等待的时间较长。我们现在来讨论远程过程调用实现中的各种问题（3.5.1 节到 3.5.6 节）。3.5.7 节将讨论在 SUN 的 ONC RPC 中是如何处理的。

3.5.1 参数类型

同本地过程调用一样，参数类型在远程过程调用中相当重要，因为它将影响到你对最后结果的期望。远程过程调用有三种基本的参数类型：

- ◆ 只输入：这种类型的参数只能把信息传送给服务器端，类似于传值调用。
- ◆ 只输出：这种类型的参数只能把信息从服务器端传送给客户端，客户端不能利用这个参数把信息传送给服务器端。
- ◆ 输入和输出：这种类型的参数能把信息传送给服务器端，服务器端也能使用同一个参数类型把信息传回给客户端，一般使用传值/结果调用（call-by-value/result）实现。

3.5.2 数据类型支持

72 数据类型支持与远程过程调用中使用的参数的数据类型相关。和许多编程语言限制参数的复杂性一样，RPC 也对其作了限制。通常 RPC 限制参数的数目，而允许更复杂的数据类型，比如限制只使用一个参数，但允许复杂的结构，这样对程序员不会有太大的不便。

3.5.3 参数整理

为了有效地进行通信，参数和更大的数据结构必须使用 RPC 整理。整理（Marshalling）是指把信息以紧凑的方式打包从而使网络上传输的信息量最少。打包的方法必须精确，这样

接收者才能够正确地解开信息。整理在每次发生 RPC 函数调用时,由所调用的桩函数执行,如图 3-9 所示。桩的使用需要库的扩展,不仅减轻了程序员负担,还有助于透明性的实现(透明性问题将在第 8 章讨论)。有些 RPC 实现提供了更多的必要的桩函数 [NIST93a]。在实际中, RPC 用于隐藏所使用的底层进程间通信的方法,如套接字(详见 3.4 节)。

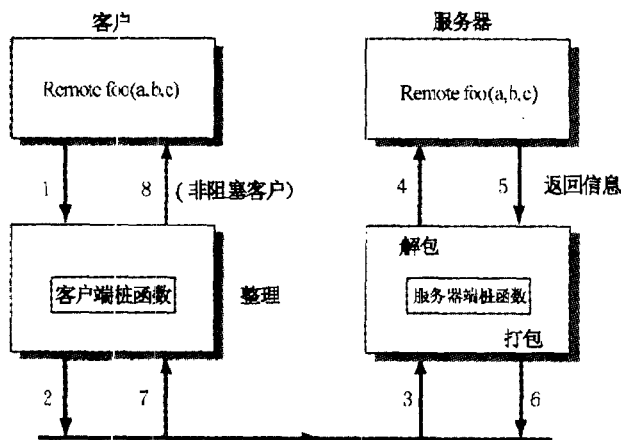


图 3-9 远程过程调用桩函数

3.5.4 RPC 绑定

在使用远程过程调用之前,客户和服务端必须建立通信。类似于套接字,在客户向服务器发送 RPC 之前,服务器必须存在且已登记。登记是向内核的端口管理者申请一个端口(如果不知道服务器,则必须与目录服务器(详见第 8 章)连接)。这样服务器端可以确定侦听哪一个端口来与客户端通信,客户端则必须与端口管理者联系并接收访问服务器的句柄,如图 3-10 所示。这个过程就是绑定并且发生于如下三个不同的时刻,它对程序员和用户是透明的。

73

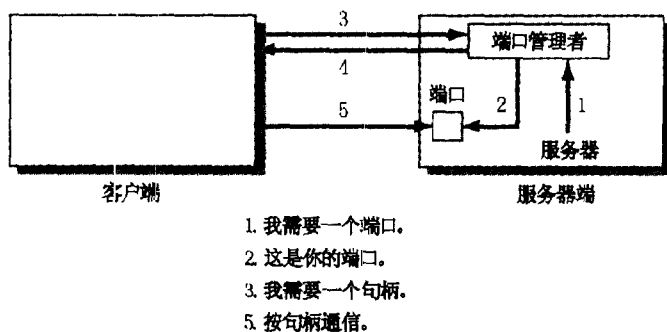


图 3-10 建立 RPC 通信

- ◆ 编译时：需要知道服务器的地址且被编译成客户的代码。这虽不是最灵活的方法，但对静态的应用程序配置来讲很方便。
- ◆ 链接时：客户端在申请服务之前请求句柄。这种情况下，将把句柄放入客户端的高速

缓存以备将来的请求。

- ◆ 运行时：从建立第一个调用开始，客户与服务器绑定。服务器发回句柄以及其他 RPC 返回值。

警告：只要服务器在运行，提供给客户的句柄就一直有效。如果服务器有几个客户并且只想取消一个客户的特权，那么它必须退出并获取一个新的端口，这个新端口必须重新与余下有效的客户进行绑定，再提供给它们一个新的句柄。总之，服务器不可以拒绝一个具有有效句柄的客户的访问。

3.5.5 RPC 认证

在分布式系统中，客户会希望验证所使用的服务器的身份。同样服务器也希望验证客户的身份。从账户中取钱的真是 Alice 吗？分布式认证和数字签名的更多信息详见第 11 章。

3.5.6 RPC 调用语义

调用语义决定重复过程调用会发生什么。由于网络出错会使返回值或者初始的过程调用丢失，也有可能破坏或延迟时间太长，导致调用的过程认为调用已经丢失。这样，过程调用就会重复。看起来没什么，但如果过程调用是从你的银行账户中取 \$300 呢？重复调用，如果系统记录你多取了 \$300（但不给你钱！），你肯定会不高兴。语义实现的精确性决定于系统使用的是有状态的还是无状态的服务器。有状态的服务器在一张表中保存了客户的状态信息，所以随后的调用利用先前调用的信息而不重传重要的信息，这也减少了网络的负载和总的传输时间，从而加快了处理请求的速度。然而如果服务器崩溃并且信息丢失，客户却无法知道。有些语义在有状态的服务器上容易实现，另外一些则容易在无状态的服务器上实现。

以下是 4 种流行的 RPC 调用语义：

最多一次

最多一次语义经常在有状态的服务器上实现，它要求确保重复请求同一个 RPC 时将不再被服务器处理。服务器通常会检查它的状态信息表来辨别重复的过程调用。

最少一次

最少一次语义确保远程过程调用至少执行一次，有可能是多次。这个语义不指明或保证哪一个过程调用将返回值，但保证将返回一些值。

多次调用的最后一次

多次调用的最后一次语义要求每一个远程过程调用的请求都包含一个标识符，客户只接受最近一次请求的返回值。

幂等

幂等语义用于无状态和有状态服务器的 RPC 应用。虽然远程过程调用能执行多次，但保证结果中没有非预期的效果。对于有状态服务器来说，这意味着客户的多次请求不能破坏服务器的状态。

3.5.7 SUN 的 ONC RPC

RPC 的第一个商业实现是 SUNTM 的开放式网络计算（Open Network Computing, ONC），RPC，最早经常被称为 SUN RPC。最初的实现高度依赖 SunOS，就是现在著名的 SolarisTM 操作

系统的前身。ONC 也有一个独立传输 RPC (Transport Independent RPC, TI RPC), 可用于不同的传输层协议和更完全的 C/S 桩函数。

ONC RPC 同时支持最多一次和幂等调用语义。此外它还提供对广播 RPC (多个接收者)、无应答或批处理 RPC 的支持。批处理 RPC 不需要返回值, 一般用于更新日志。由于 ONC 只允许两个参数: 一个输入参数和一个输出参数。在参数类型实现方面看, ONC 是有限制的, 但是它支持几乎所有的 C 语言中的数量和集合数据类型, 因而用户可以把需要的参数组合成结构并把它传送给远程过程。ONC RPC 提供三个层次的身份验证, 使用数据加密标准 (Data Encryption Standard, DES) 的最高层称为安全 RPC, DES 详见第 11 章。

3.6 小结

进程间通信是多进程完成同一任务中的一个关键问题, 程序员可以使用各种技术。表 3-1 根据分布式系统中能使用的高级部件的类型总结了本章所描述的方法。第 12 章的案例学习会更进一步地讨论进程间通信。当使用进程间通信时, 用户必须知道每一种方法的优缺点。消息传输可以以阻塞或非阻塞的方式用于分布式或并行系统中。管道使用缓冲, 也提供同步, 只要进程能共享目录空间就可使用, 但程序员必须意识到这会约束应用的扩展和应用的规模。通常为避免管道的这种严厉的限制, 最好使用套接字。从套接字为程序员提供的控制水平上来看, 它可能是功能最强大的进程间通信机制了, 但这些增加的控制也带来了许多处理附加细节的麻烦。除非用户富有经验并能熟练开发, 否则套接字还是很麻烦的。在详细说明 3.4 和 3.5 中, Java 通过类定义对套接字的支持给予了许多允诺, 但只有时间才能证明这一点。远程过程调用在分布式系统中很有用, 但因为缺少桩函数支持, 所以需要程序员去实现桩函数 [NIST93a]。我们还要记得在 RPC 下层通常会发现我们的朋友——套接字。为提高这个多进程间计算的关键方面的能力, 在该领域的研究和发展仍在继续。

76

表 3-1 IPC 机制总结及其在分布式系统组件中的应用

IPC 方法	分布式部件	实时部件	并行部件
消息传输	是	是	是
管道	否	是	是
套接字	是	是	是
RPC	是	是	是

3.7 参考文献

以下参考资料提供了进程间通信及其实现的一般信息 [Blo92, Cha97, Fla96, Gra97, Lap92, NiPe96, RoRo96 和 QuSh95]。部分与本章信息相关的传统研究论文包括 [BaHa87, BALL90, BeFe91, BiJo87, Bir85, BLPv93, Cla85, Fer95, GaSp91, Gen81, Gib87, GiGl88, HuPe91, KiPu95, KJAKL93, NeBi84, NIST93a, RaSh92, ScBu89, SrPa82, Sun88, TaAn90 和 WuSa93]。

下面提供的是一些关于进程间通信和同步内容在 Internet 上的资源, Winsock2 的信息在 <http://www.intel.com/ial/winsock2>, 1.1 版本的 Windows Socket 的常见问题 (FAQ) 在 <ftp://SunSite.UNC.EDU/pub/micro/pc-stuff/ms-windows/winsock/FAQ>。一个全面的 Winsock FAQ 的网址

是 <http://www.well.com/user.nac/alt-winsock-faq.html>。[NIST93a] 的在线版本在 <http://nemo.ncsl.nist.gov/nistir/5277/>。<http://ei.cs.vt.edu/~cs5204/rpc/rfc1057.html> 是 [Sun88] 的在线版本。

习题

- 3.1 就 3.1 节给出的选择因素（不包括第一个因素）评价本章中的 IPC 机制。
- 3.2 描述一个应用情景，在这个情景中，程序需要同步通信。
- 3.3 描述如何使用非阻塞原语模拟阻塞消息传送。
- 3.4 描述如何使用阻塞原语模拟非阻塞消息传送。
- 3.5 描述一个应用情景，在这个情景中，程序需要异步通信。
- 3.6 描述阻塞相对于非阻塞消息传送的优缺点。
- 3.7 分别描述以下消息传送编址类别的一个用途，
 - a. 一对一编址
 - b. 一对多编址
 - c. 多对一编址
 - d. 多对多编址
- 3.8 分别描述一个方案来实现以下消息传送编址，
 - a. 一对一编址
 - b. 一对多编址
 - c. 多对一编址
 - d. 多对多编址
- 3.9 描述一个情景，在这个情景中多对多消息传送的随机排序会产生问题。
- 3.10 写一个客户端和服务端共享 UNIX 命名管道的程序。客户端程序接受用户的输入并写入命名管道，服务端程序从命名管道中读取信息并显示在屏幕上。
- 3.11 讨论套接字和管道间的相对优缺点。
- 3.12 Blackjack 是一个流行的纸牌游戏，一副纸牌包括 52 张，有 4 组：红心、方块、黑桃、梅花。每组有一张 A、1、2、3、4、5、6、7、8、9、10、J、Q 和 K。J、Q 和 K 被认为是花牌，点数按如下的方式规定：

花牌：10 点，

A 到 10：分别是 1 到 10 点。

发牌人先给每一位游戏者一张牌，然后再给第二张，接着游戏者可以决定是否再要牌，除非 a. 要求停止，或 b. 总数超过 21 点，否则游戏者可以任意要牌。

考虑只有一个发牌人，游戏者最多只有 5 个。为简化起见，我们只允许 A 等于 1 点，不允许发牌人参与游戏。

按以下的规则决定赢者：

 - a. 第一位不超过但达到 21 点的游戏者。
 - b. 不超过但最接近 21 点的游戏者。
 - c. 不超过但最接近 21 点的点数相同的一组游戏者。

游戏者可以允许玩多次，但每次都重新洗牌（提示：使用随机数字产生器），游

戏者在加入一个新游戏时有时间限制。

选项 1：使用命名管道实现游戏。

选项 2：使用 UNIX 套接字实现游戏。

选项 3：使用 Internet 数据报套接字实现游戏。

选项 4：使用 Internet 流套接字实现游戏。

选项 5：实现选项 3 和选项 4。

- 3.13 一个轻量级的 RPC 允许在同一主机系统中进行 RPC 调用，描述其优点。
- 3.14 描述一个得益于非阻塞 RPC 的应用。
- 3.15 ONC RPC 只允许有两个参数：一个输入参数和一个输出参数，其潜在的优点是
什么？

第4章 内存管理

正如第2章中所提到的，内存管理通常是内核职责的一部分。在所有系统中它都是一个关键因素，因为所有执行的程序及其数据都需要驻留在内存。内存管理往往不仅依赖于操作系统，而且依赖于底层的机器结构。该领域的一些最新研究涉及如何将内存管理从计算机架构中分离出来。

在本章中，4.1节首先简要回顾一下有关集中式内存管理。若需要进一步了解相关知识，请参阅你喜欢的有关操作系统的入门书籍，然后我们进一步讨论高级操作系统中的内存管理。内存分为几部分：高速缓存、主存（随机存储器，RAM）和辅存。辅存通常在磁带、硬/软盘驱动器、CD-ROM或DVD上实现。高速缓存（包括闪存）是最快（也是最贵）的内存，但是它是易失性的，存储在里面的数据不允许掉电。直接配在CPU上的高速缓存比通过总线（通常在主板上）连接的高速缓存要快。主存是第二快速的内存，也是易失性的。辅存速度较慢且便宜，但它是非易失性的。搜索、访问和传输速度极大地依赖于存储的种类，并在不断提高。

81

一个具有单一存储空间的并行计算机将利用4.2节中提出的简单的存储模型。4.3节描述的使用NUMA结构的共享内存模型也常常被采用，小规模分布式系统也采用这种模型。另外，分布式系统可能利用的扩展虚拟存储和分布式共享内存模型将在4.4节讨论。最后在4.5节中讨论分布式存储管理中尤其重要的存储迁移的实现问题。

4.1 集中式内存管理回顾

为了理解高级操作系统，有必要了解一下集中式内存管理。具体地说，分布式系统中分布式内存管理器驻留在集中式内存管理器顶端是很常见的。真实内存指实际的物理主存，大多数集中式系统不仅使用物理主存，还提供虚拟内存。本节首先简要回顾一下虚拟内存（4.1.1节）、页面和段（4.1.2节）、页面替换算法（4.1.3节）的基本概念。

4.1.1 虚拟内存

虚拟内存在现代的集中式系统中很普遍，它使进程能访问的内存超过实际可用的数量（不论是对系统还是进程）。不在主存上的虚拟内存位于磁盘内，内存管理器在系统的虚拟地址空间范围内为进程提供虚拟地址，并负责管理虚拟内存的哪部分位于物理内存，哪部分位于磁盘。虚拟内存到物理内存或磁盘的映射由内存管理单元（memory management unit, MMU）实现。

4.1.2 页面和段

为移动和编址的方便，内存经常被分成段或页面。段根据数据的大小而有所改变，但总是为最接近的2的整数幂。页面的大小一般相同，内存的页面在物理内存中放置成与页面一样大小相同的块。浪费的内存空间有两种：内部碎片和外部碎片。内部碎片是一个内存块中浪费的空间，而外部碎片是主存中不同数据块之间所浪费的空间。对于一个系统来说有

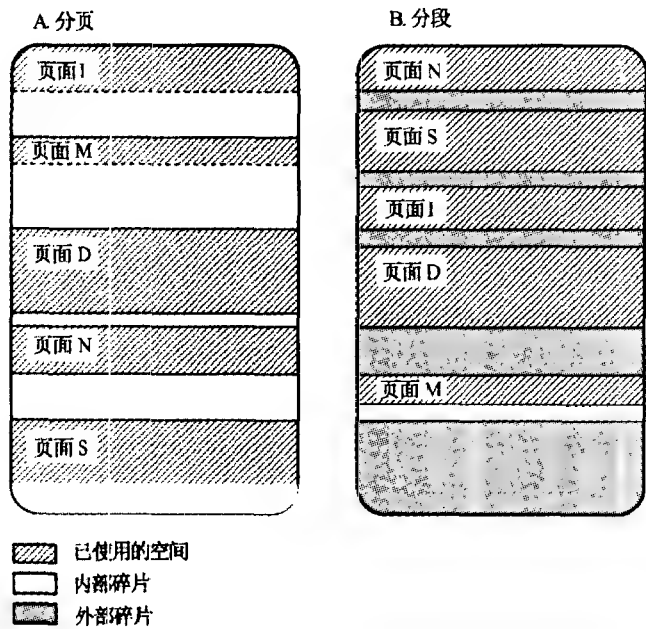


图 4-1 基于分页的内存和基于分段的内存中的碎片

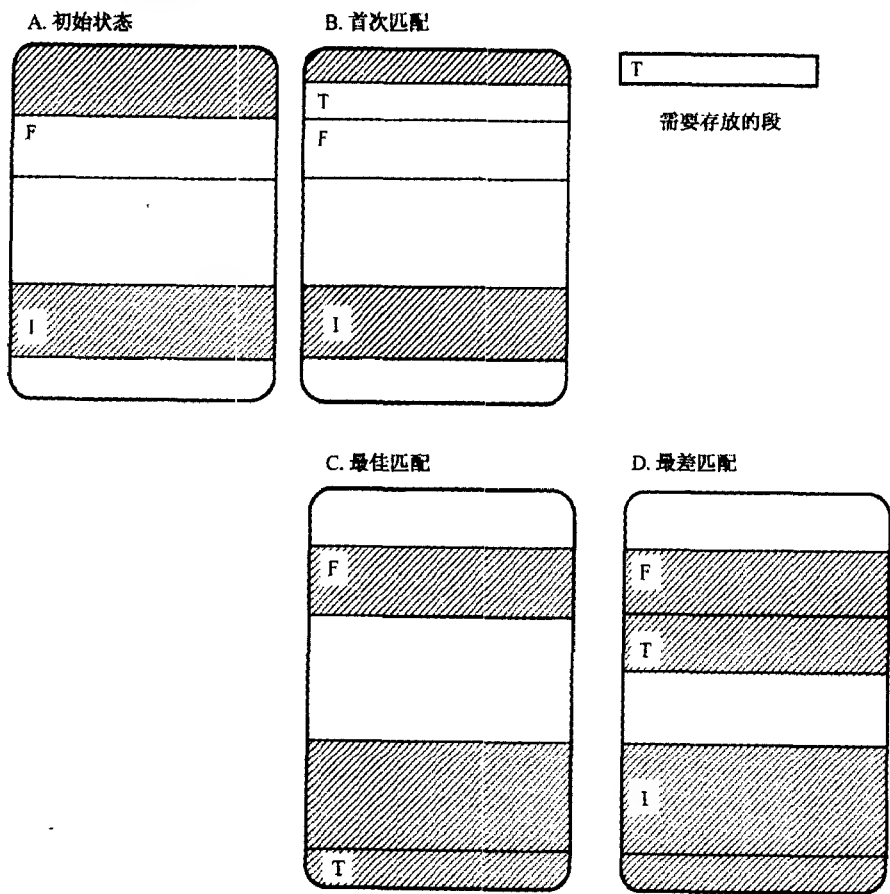


图 4-2 选择段存放位置的算法

30%的内存由碎片所浪费是司空见惯的。图 4-1 显示了基于分页的内存和基于分段的内存的内部和外部碎片。

内存段深受外部碎片和偶尔少量的内部碎片影响。外部碎片是内存中段的移入移出操作所造成的。初始的段可能被分成适当的大小并且紧密排列，当一个段移出物理内存，新移入的段不是同样大小的概率非常高，这就会在段与段之间产生“空洞”。随着动态地连续不断地将段移入移出物理内存，就会导致许多空洞的存在。

这些空洞就是外部碎片。选择一个空洞用来存放一个段的通常算法是首次匹配、最佳匹配和最差匹配。首次匹配是将段放在它发现的第一个可行的洞中，最佳匹配是将段放在一个尽可能小的洞中，最差匹配是将段放在尽可能大且足以容纳该段的洞中。这些算法在图 4-2 中进行了描述。

因为所有的页大小相同，但一个进程所需的内存可能是变化的，所以分页会受内部碎片影响。然而统一的页面大小消除了外部碎片。当一个页面从物理内存中移走时，所留下的“空洞”刚好是另一个页的大小。页面通常用在虚拟内存系统中，用来管理页面信息的数据结构是一张分页表。这张表存放着不同页面的虚拟地址（尤其是一个虚拟页面号和页内偏移量）。虚拟页面号是分页表中一个简单的索引。

4.1.3 页替换算法

当一个进程需要一个没有驻留在物理内存中的页面时会产生一个**页面错误**。页面替换算法是用来决定物理内存中哪个页面将被替换出去，用来存放所需要的新的页面。这个替换算法可能适用于整个系统的物理内存，分配给导致页面错误的同一用户的内存或导致页面错误的进程的物理内存。因移动页面需要耗费资源，所以这是一个重要的决策，页面替换算法的首要目标就是做出正确的选择。理想的正确选择就是任何选中替换的页面不再使用；而最坏的选择是选中替换的页面即将被访问，这会导致另一个页面错误！如果系统仍然不断地读取和再读取某页面将导致系统性能受到严重影响，这就是**颠簸**（thrashing）。一方面颠簸是糟糕的页面替换算法的结果，通常也是多道程序设计程度相对于可利用资源太高而造成的。如果系统要减少多道程序的道数，将会减少需要服务的进程的数目。反之，会增加每个进程可用的物理内存及减少页面错误的数目。糟糕的编程技术也会导致不必要的页面错误（比如，以列为主序访问数组，但编程存储时却以行为主序）。

第一个页面替换算法是先进先出（FIFO）。在这个算法中，根据页面开始进入内存的时间进行替换。当发生页面错误且没有可用的帧的时候，最先进入内存的页面将最先被移出。这个算法的缺点是最先移入的页面通常也是最重要的页面，会被频繁地调用，很可能马上就要再次被调用。第二个页面替换算法是非最近使用（NRU）。这个算法是放弃最近不被使用的页面。最近可以是静态阈值，可以使几个页面具有同等的放弃候选资格。第三个页面替换算法类似于 NRU，是最近最少使用。这个算法移出最长时间不被访问的页面，认为是不再需要。设想一个装满了衣服的衣柜，不能再多装一件，如果买了一件新外套想把它放入衣柜，最大的可能选择就是拿走一两年都不曾穿过的衣服，而不是拿走上周刚穿过的外套。这是这个算法的基础，这也是一个流行的算法。这个算法的衍变有第二次机会算法。第二次机会算法在页面被选择移出的时候第一次对其进行标记，除非页面已经被标记过，否则不移出。当一个内存页面被访问时则清除标记（如果有标记），页面替换算法则继续寻找符合标

准及标记过的页面。其他算法会考虑到移出页面的费用。这种类型的“懒惰”算法努力去寻找不需要重新写回磁盘的页面，并通过跟踪含有被改写过数据的页面（脏页面）来实现。扔掉一个干净的页面总比扔掉一个脏页面来得快。

4.2 简单内存模型

简单内存模型用于并行 UMA 系统的内存管理。在这个简单内存模型中，所有处理器的访问次数都是相等的，因此所关心的主要问题是全局内存的并行程度。我们讲过集中式系统中多道程序程度太高时，没有一个单独的任务有足够的内存空间来执行。所以每个任务不断地调用没有载入内存的页面，并产生颠簸，这会导致很大的开销。同样，并行系统必须提供给处理器足够的内存。当然，在经济上可行的时候，大量内存解决方案更具有吸引力，可以利用不太昂贵的内存。

高性能并行系统经常选择不使用虚拟内存和高速缓存。虚拟内存需要额外的管理负担，这些负担会减低以最佳性能为设计和需求目标的系统的性能。鉴于高性能系统中实现程序的类型需计算的数据量很大，这样需要大量的高速缓存。另外，因为这些数据通常不会重用，高速缓存的优点越发减弱，所以高速缓存也变得越发不实用了。

4.3 共享内存模型

当在一个小的分布式或多处理器（NUMA）环境中计算时，就会用到消息传输模型或**共享内存模型**。像第3章所提到的那样，共享内存模型是为完成进程间通信的附加模型。实际上，共享内存因其能快速完成任务（如在调制解调器和拨号路由器的网络接口之间传递消息）而成为最流行的方法。共享内存模型把虚拟内存的概念扩展到多物理内存、本地的高速缓存和辅存。共享内存模型在使用消息传输（或远程过程调用）时会遇到困难，因其不是使用大而复杂的数据集合的有效的方法。共享内存相对消息传输的最大优点是容易数据分割及通过迁移执行载入分发。共享内存模型允许多个进程读、写其公共内存中的数据结构，这样系统必须引入一些并发控制（见第5章）和事务管理（见第9章）。共享内存模型的另外一个优点是共享同一内存的含有本地高速缓存的工作站的使用，这些工作站再通过总线连接来形成共同的共享内存。这方面对于多处理器和本地分布式系统非常有用，因为它在保证经济负担最小的情况下，大幅度地提高了性能和计算能力。但也面临这样的困难，即通过单一总线不断地访问内存，也就是说，工作站会经常面临总线争夺而导致瓶颈，所以一般不建议在单一总线上使用超过32个处理器的时候使用这种方法。详细说明4.1描述了Amoeba内存管理系统，详细说明4.2则描述支持共享内存的UNIX系统V。

85

86

详细说明 4.1

AMOEBE

Amoeba 分布式操作系统的微内核在 [TaKa94] 中有相关描述，它包含内存管理系统。这个内存管理系统以功能性、简洁性及经济性为目标进行设计，它要求整个进程驻留在内存中以提高 RPC 操作的速度，而不限制一个给定的进程所使用的段的数目。从某一段开始的一个进程可以创建和破坏在它运行时所使用的段。当一个进程创建一个段时，它就有能力使用这个段。这里使用硬件的内存管理单元（MMU），段不被交换或分页，但

能位于虚拟内存的任何位置，而且必须全部位于虚拟内存中，否则就不能使用 Amoeba 执行。这些段可以由同一系统中的各种进程共享，但不能通过网络共享。

详细说明 4.2

支持共享内存的 UNIX 系统 V

UNIX 系统 V 把共享内存作为进程间通信的工具进行支持。为了能在 C 或 C++ 程序中使用这个共享内存支持，必须使用如下的包含语句：

```
#include <sys/shm.h>
```

为创建共享内存的这个功能调用叫 `shmget`，函数 `shmat` 和 `shmdt` 追加和分离共享内存到用户的内存空间。所有细节的完全列表在本地系统（`man shmget`，`man shmat` 或 `man shmdt` 类型）的手册（在线 UNIX 手册）中都有提供。另外，UNIX 系统对共享内存的所有支持的列表都能通过关键字索引选项（`man-k shared memory` 类型）找到。

4.3.1 共享内存性能

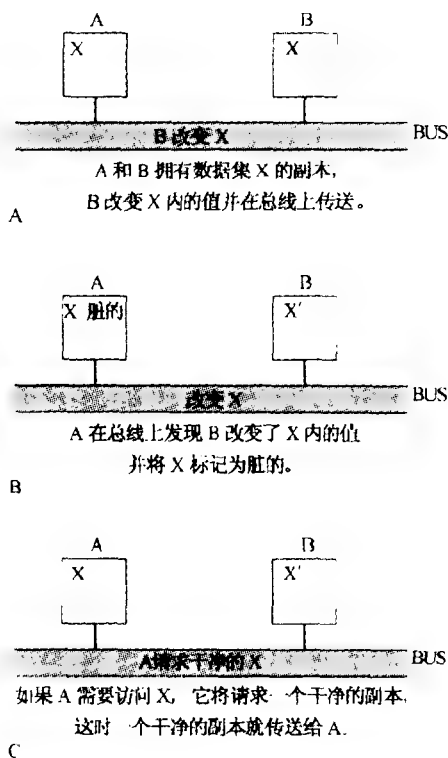
随着处理器速度的不断提高，内存的瓶颈日益严重，系统在等待从内存中检索信息的时候花费了几乎所有的空闲时间。系统通常标榜它们的性能，但这个性能是指性能的峰值，而不是平均性能。一些阻止完全利用快速处理器和许多现有资源的最大的问题就是内存速度、内存访问和通信延迟，这些同样存在于共享内存系统中。使用共享内存的一个最基本的问题是规模——如果处理器的数目和内存的数量增加了，系统如何更好地保证它的高效性能。当增加并行或分布式系统的内存和处理能力时，可能会认为性能也在线性增加。不幸的是，有句老话“厨师多了煮坏汤”在这里呈现全新的含义。通常，不在于有厨师数目的多少，而在于他们之间的协调和利用！

性能对所有的应用都很重要，尤其是对于实时应用。通常超级计算机和实时应用不使用任何类型的虚拟内存，正如俗语所说“如果不是真实的（意思是虚拟的），那么也就不是快速的”。对于实时要求不太严格的应用，Sun 和 Zhu 提出了两个能提高共享虚拟内存性能的因素 [SuZh96]。首先，系统应该让通信与计算交叠，一个不允许交叠的系统在效率方面不可能同提供通信和计算交叠的系统一样高。其次，系统应该尽可能地自动预读取数据。如果读取远程数据的指令能够在系统需要内存中的数据的前一步执行，那么至少可以减少等待远程数据的延迟或使延迟最小。软件开发技术允许提前辨别这些信息，这个辨别过程可以在运行时间或接近运行时间由编译器自动执行。

另外还有两个因素可能会影响共享内存系统的性能。首先回忆在 NUMA 构架（根据定义）中，非本地内存引用比本地的内存引用要昂贵得多。另外按照惯例，非本地内存引用与本地的内存引用费用比例可高达 10:1，所以可以通过监视非本地内存引用来提高性能。当达到某一阈值时，就使用算法把这一页迁移到本地内存中去。这个任务一般由一个叫页扫描器的系统守护进程执行，它会监视适当的统计数据并迁移页面。如果系统是一个同类系统，迁移仅作为页面错误处理，4.5 节将讨论迁移的相关问题。

4.3.2 高速缓存一致性

如果必须要预读取数据,那么共享内存成功实现的一个重要因素就是高速缓存应保持一致性。一致性是用于描述确保所有复制数据相同并且正确的功能术语,有三种实现高速缓存一致性的方法。第一种是利用软件加强临界区,即某一给定进程将改变共享数据时所处的受保护的代码区域,将在第5章详细讨论。第二种是利用软件阻止处理器把共享内存作为高速缓存。第三种方法是指窃听式高速缓存,如图4-3所示。在第三种方法中,所有的处理器都通过共享总线相连,每一个处理器不断地窃听或监视共享总线。如果某一个处理器探测到总线上的信息将要改变本地已缓存的数据,那么就把含缓存数据的那一块标记为“脏的”;如果处理器要访问脏的高速缓存,它必须从共享内存系统中请求一个干净的副本。窃听式高速缓存确实很大程度地减少了总线流量,但对增加系统处理器的最大数目却没有什帮助。各种窃听式高速缓存在使用多总线的系统中也有应用,只是算法比保持高速缓存一致性的进程更复杂。关于各种类型的数据一致性的内容将在第9章论述。



88

图 4-3 窃听式高速缓存

4.4 分布式共享内存

分布式共享内存模型最先是由 Li 和 Hudak 在 1989 年引入的 [LiHu89], 并且从那时起在处理上引起了广泛的讨论。一个使用分布式共享内存 (DSM) 的系统, 通常认为是多计算机系统, 由具有本地内存的独立的处理器组成。这些处理器通过互连网络相联, 这一点在第 1 章中已有讨论。消息传输和分布式共享内存主要的区别是 DSM 给出了共享内存的逻辑视图, 因而对应用来说, DSM 系统实现了必要的通信并保持了应用程序中数据的一致性。像共享内存模型一样, 在 DSM 中多数是为了提高效率而保留了本地的高速缓存, 本地内存的部分或全部都映射到 DSM 全局地址空间中。DSM 系统为驻留在系统中的所有数据维护一个目录服务, 用来存储完整的状态和位置信息。有多种实现这种目录服务的模型, 但使用的方法应由系统需要的一致性类型决定 (详见第 8 章的分布式目录和第 9 章中有关各种类型一致性模型的讨论)。无论实际的内部目录是如何实现的, DSM 系统都通过利用单个目录或几个分布式目录来维护完整的信息。

89

4.4.1 分布式共享数据的方法

为了管理共享数据, 必须做好下面的决策:

- ◆ 共享数据如何分布？
- ◆ 一个给定数据集允许多少读者和写者？

有两种方法分布共享数据：复制和迁移。复制会保留多个副本，每一个需要数据的地方都有自己的本地副本。迁移则把数据的副本移动到需要它的不同的位置。使用迁移只有一个地方允许有副本，如果另一个位置需要使用这些数据，除非当前的数据持有者放弃，否则任何请求都不予以满足。如果 DSM 系统收到多个请求，它将根据请求的顺序或优先级来决定哪个请求得到批准。

管理共享数据还涉及到决定允许并发访问某一数据集的读者和写者的数目。允许多写者的系统比把写者数量限制为一个的系统要复杂得多。以下介绍三种方法：单读者/单写者、多读者/单写者和多读者/多写者。

单读者/单写者

顾名思义，这种类型的方法对某一特定数据集只允许一个读者和一个写者，因而禁止并发使用共享信息且不允许复制。解决方案可以是集中式的也可以是分布式的。使用集中式解决方案，DSM 系统需要一个集中式的服务器，潜在的瓶颈将会是一个关键问题。分布式解决方案一般需要一个静态的数据分布，因而需要一个进程联系两个不同的地方以获取不同的数据集，但是同一个数据集通常由同一个位置控制。如果分布式的解决方案不允许迁移，那么某一给定数据集不仅由同一位置控制，而且还要一直位于这个位置。允许迁移的单读者/单写者分布式 DSM 算法也称为“热土豆”算法 [KeLi89]。

多读者/单写者

顾名思义，这种类型的方法允许多读者但只允许写入一个惟一的位置。当多读者都拥有一个数据集的副本且单一的写者改变了这个数据，那么读者的副本就不再准确。为了避免这种情况，大多数实现都对拥有只读副本的位置给出无效处理。这种无效处理需要一个副本集合，即系统中拥有每一份数据副本的成员的列表。多读者/单写者方法可以由集中式的解决方案、集中式和分布式解决方案的组合或分布式解决方案来实现。

集中式解决方案需要一个服务器（关键元素）处理所有的请求并维护所有的数据，同时也负责发送所有的无效通知，如图 4-4 所示。我们已经介绍了每一种集中式的解决方案，知道它并不是最有效的解决方案，因为它增加了服务器周围的网络流量，同时又需要服务器超额工作。

另一种可选择的集中式解决方案的方法包含了部分分布状态。具体地说，集中式服务器接收请求信息，所有的数据静态地分布在整个系统中。集中式服务器接收到请求后就把信息传送到负责某一特定信息的分布位置，每一位置负责处理集中式服务器转发的请求，并在数

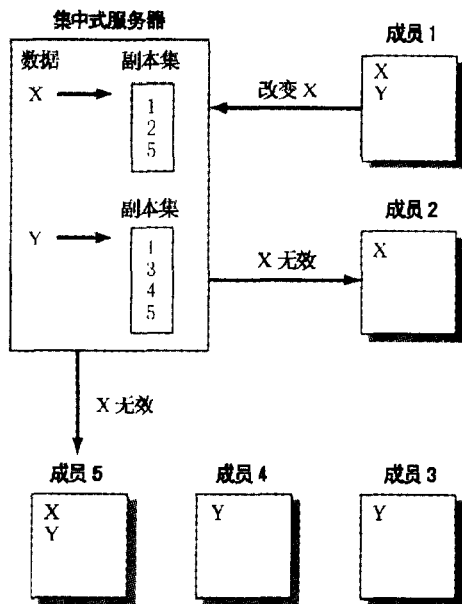


图 4-4 多读者/单写者 DSM 的集中式服务器

据无效时通知读者，图 4-5 描述了这种实现中的一个无效的范例。

对 DSM 的多读者/单写者方法来说有两种基本的分布式方法。第一个分布式方法是动态的，信息可以改变位置。为了定位数据集合的拥有者，所有的请求都在系统中进行广播，如图 4-6 所示。

这个方法增加了系统中传输的信息量，因而对整个分布式系统产生了更多的额外负担。详细说明 4.3 给出了动态算法。第二个方法是静态地分布数据。数据的拥有者（如持有特定数据集合的位置）负责接收和处理所有与这些数据相关的信息并在数据无效时通知读者。因这种方法涉及静态分布，所以每一个成员都知道每一个数据存储在哪里。详细说明 4.4 给出了静态算法。这种解

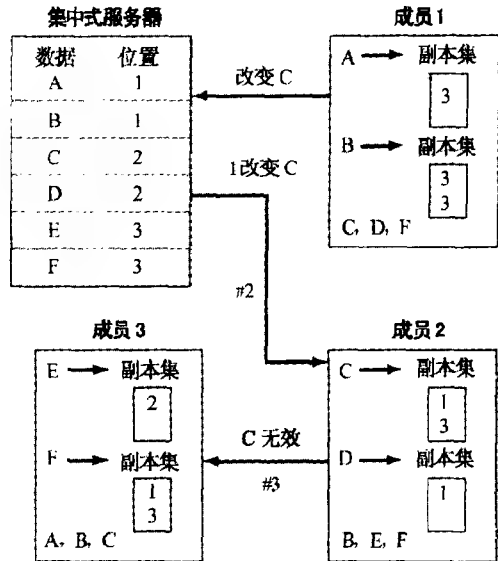


图 4-5 多读者/单写者 DSM 部分分布的无效方案

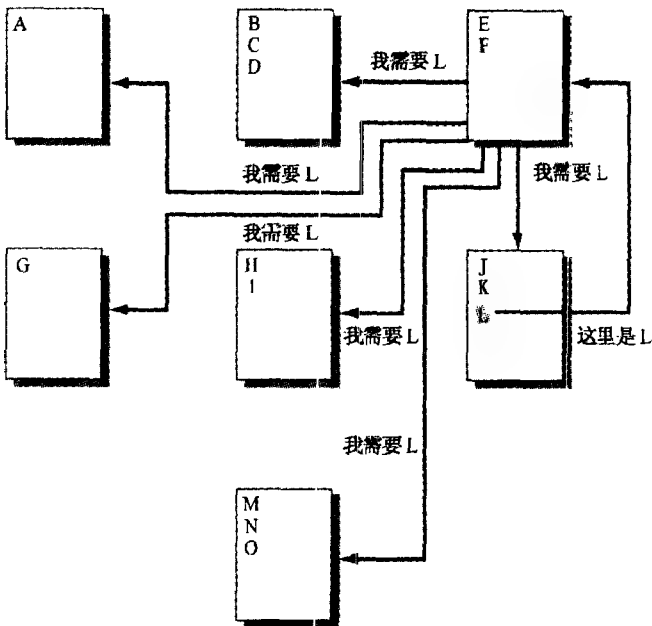


图 4-6 动态分布式多读者/单写者 DSM

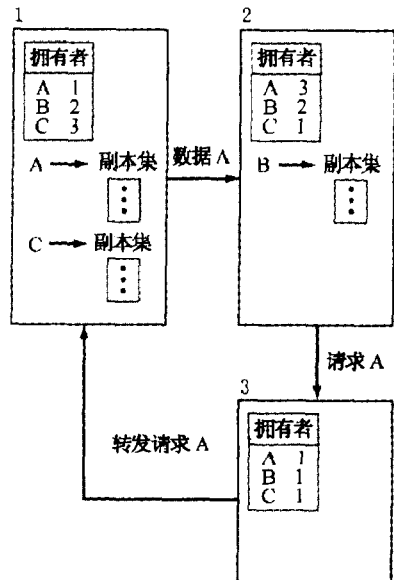


图 4-7 多读者/单写者 DSM 的动态数据分配

决方案的一个变通之处是允许动态地分配数据，把请求传送给数据的可能拥有者或最后拥有者。如果是真正的拥有者，那么问题就解决了；如果数据已重新分配过，请求的接收者就会把请求转发给它认为的数据持有者，如图 4-7 所示。详细说明 4.5 描述了 Chorus 中的 DSM 系统。

详细说明 4.3

动态多读者/单写者 DSM 的分布式算法

动态分布式请求页面

```
Broadcast Request Page(Page Number);
```

动态服务请求页面

```

If receive Broadcast Request Page (Page Number,
    read write status) & Owner(Page Number);
    // 我们拥有被请求的数据
Then
    {
    If write(Page Number) < 0;
        // 如果页面没有当前写者
    Then
    {
    return(Page Number, read write status);
        // 当满足单写者的条件, 同时处于
        // 请求的读 / 写状态时, 释放页面

        If read write status = write
            // 如果请求写
        Then write(Page Number) = 1;
            // 更新页面的写标志
        Since Add to copy set (requester);
            // 增加到本集的读者
        // 没有当前写者终止
    }
    Else
        // 当前有一个写者
    {
    Wait for writer release;
    return(Page Number, read write status);
        If read write status = write
        Then write(Page Number) = 1;
        Since Add to copy set (requester);
    }
    }
    // 有一个当前写者终止
    // 已收到请求终止
}

```

动态无效方案

```

If receive(change, page Number)
Then forall in copyset (page number)
    send invalid(page number);

```

详细说明 4.4

静态多读者/单写者 DSM 的分布式算法

静态分布式请求页面

```

Owner-look up owner(page number);
Send (owner, Request Page (Page Number));

```

静态服务请求页面

```

If write(Page Number) < 0;
    // 如果页面没有当前写者
Then
    {

```

```

return(Page Number, read write status);
// 当满足单个写者的条件, 同时处于
// 请求的读/写状态时, 释放页面

If read write status=write
// 如果请求写
Then write(Page Number)=1;
// 更新页面的写标志
Else Add to copy set (requester);
// 增加副本集的读者
// 没有当前写者终止
// 当前有一个写者
Else
{
Wait for writer release;
return(Page Number, read write status);
If read write status=write
Then write(Page Number)=1;
Else Add to copy set (requester);
}
// 有一个当前写者终止

```

静态无效方案

```

If receive(change, page Number)
Then forall in copyset(page number)
Send invalid(page number);

```

详细说明 4.5

CHORUS

Chorus 是 1980 年开始研究的一个分布式实时操作系统项目, 在 1997 年底开发到第 3 版, 成为商业产品并由 Sun Microsystems 购买。从此, Chorus 成为 Sun 的 JavaOS 的基础并以 ChorusOS 的名称由 Sun 销售。

Chorus 使用动态分布式 DSM 算法, 共享内存的单位是段, 一个段可以由系统的几个成员占用, 用于只读访问, 或者由单个位置占有, 用于读写访问 (多读者/单写者)。内存管理在内核中执行, 当一个页面发生错误时, 内核检查它的本地高速缓存。如果在本地没有发现这个页面, 它就与映射管理器联系以确认谁持有这个页面; 然后与拥有者接触, 如果拥有者不使用包含请求页面的段, 它就更新内存管理单元, 调整页面表并发送页面。当读取页面时, 引起页面错误的进程就被中断; 最后映射管理器请求内核返回“脏的”页面, 从而更新它的高速缓存。一旦高速缓存得到更新, 就可决定页面是否写回磁盘或是留在高速缓存中。关于 Chorus 的更多信息可参考 [AA092, AGGL86, ARS89, BG-GLOR94, 和 LJP93]。

多读者/多写者

为了更好地理解 DSM 中多读者/多写者方法的复杂性, 设想自己是一个项目组中的一员。如果每个人用磁盘把程序的副本带回家, 并独自修改, 那么如何自动地整合每一个人的修改? 这也就是这种类型的 DSM 所面临的问题。解决方案需要一个序列器负责给所有的读请求和写请求按时间排序, 分布式环境中的时间排序算法将在第 10 章中介绍。序列器接收

到请求并附上序列信息后，请求就广播到拥有数据集合副本的站点，广播的接收者根据序列器标识的顺序更新自己的数据副本集。序列器算法可以是集中式的或分布式的，多读者/多写者 DSM 算法也称为全复制算法。

4.4.2 DSM 性能问题

对于所有的系统，每个人都希望高效率地实现。这一点我们也一直提到，实时系统的效率不仅是所要求的，而且是必需的。具体地说，主要是要考虑颠簸、数据共享、块大小的选择和 DSM 系统的实现位置等问题。

我们回顾术语**颠簸**，即在更标准的操作系统中多道程序设计程度太高以致系统被迫不断地改变内存中的页面，因而降低了性能。在分布式操作系统中，许多位置都希望改变共同的数据集合时，颠簸是指 DSM。这就导致数据集合不断地在各个位置间传输，就像在网络上为控制数据集合而进行拔河一样。如果数据集合在某一位置反复传输，就不能对这些数据继续进行操作，性能也就随之下降，颠簸则更加严重。如果“拔河”中涉及一个页面，但在这个页面中不含有共享数据结构，则称之为**伪共享**。颠簸的一个解决方案是保证每一位位置都会在一个最小持续时间量的过程中持有数据集合，这称为暂时停顿页面。

颠簸的最佳解决方案是将适当大小的块用于数据迁移。如果块太大，多位置请求同一个数据块的可能性会增加。尺寸大的块还会导致本应单独存放的两个或更多个不相关数据存放在同一块中，这样又增加了多个位置请求同一块的可能性。当然，如果块太小，则服务器会有更多的块，那么目录服务器管理数据的费用必定增加，状态信息表的条目也会增多。因每个数据块必须单独请求并传送到要求的位置，所以大的块减少了费用量。

随之而来的问题是，从效率角度来看多大的块为最好。一般最好的折衷是在 DSM 系统中使数据块的尺寸与底层系统的页面尺寸相同，这样允许 DSM 在现存的系统顶层，利用相同的页面错误和访问控制机制，很容易地进行集成。这些具有页面尺寸的块应能确保某一给定位置在固定的最小时间内进行计算。另一方面，因分布式系统驻留在网络的顶层，如果块及页面的尺寸能与下面网络使用传输信息的包的大小联系起来，无疑可以改善性能。这个问题对应用开发者应该透明，但是如果忽略了它，分布式实时应用会自行破坏，这也是系统程序员应该注意的。

另外一个性能问题是内存等待状态和谨慎选择数据元素位置的问题。这个问题在并行结构的单内存模型中很容易解决，但在分布式环境中不可忽略。只要存在某种形式的同步，就可以辨别数据访问模式，并可能进行优化以使内存访问的延迟最小。随着处理器速度的快速增长，实时应用中这个问题越来越显著。关于这个主题的讨论可在 [ErCo92, ErCo93a, ErCo93b, Erick93b, GaCo95] 中找到。

最后一个效率问题是 DSM 应在哪里实现：算法可在硬件中、软件中或二者结合来实现。一般较昂贵和复杂的计算机使用硬件实现，也是最有效的实现。不太昂贵的系统如个人计算机在软件中实现 DSM，而工作站使用软、硬件结合的方式来实现。

4.5 内存迁移

内存管理的一项必要任务是虚拟内存移动，或叫内存迁移。内存迁移是第 2 章讨论的进程迁移中最费时的部分。对于内存迁移需要做两个基本的决策：

◆ 在进程迁移的什么时候迁移内存？

◆ 应该迁移多少内存？

下面讨论实现内存迁移的三个基本方法。

第一个方法是**停止和复制** (stop-and-copy)，这是最简单最直接的方法，但同时它也是效率最低的解决方案。当在迁移进程中使用这种内存迁移的时候，原始位置进程的执行立即被挂起，然后与这个进程相关的所有虚拟内存转移到迁移目的地。一旦与进程有关的其他信息迁移结束，进程继续执行。这个方法尤其对交互和实时应用缺乏吸引力，这样的环境根本不能接受彻底中断进程执行（也称**停顿时间** [Esk89]）直到所有的迁移完成所导致的无效率，以及转移所有相关虚拟内存时浪费的时间。图 4-8 描述了停止和复制方法。

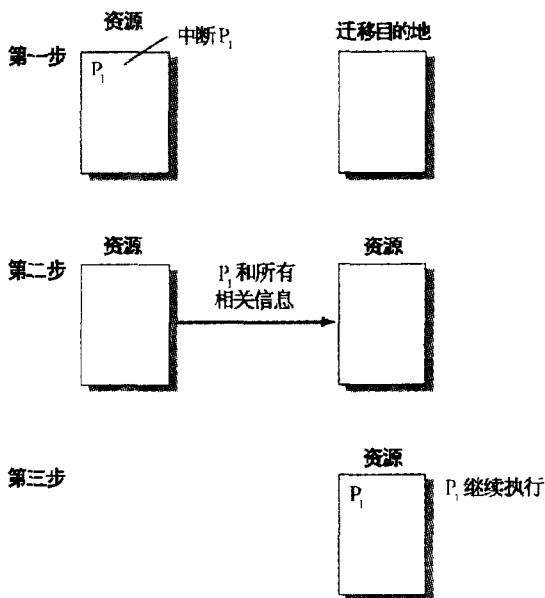


图 4-8 停止和复制内存迁移

第二个方法是**并发复制**，如图 4-9 所示。这个方法试图在迁移进程中减少因完全中断进程执行而导致的无效率。因此，当要迁移进程时，进程继续在最初的位置执行，同时并发拷贝所有的虚拟内存和其他与迁移进程相关的信息到目的地。随着进程在原始位置的继续执行，一些已迁移的页面成为脏的页面，也就是说因程序的执行，这些页面与迁移的页面所包含的信息已不相同。这样这些脏页面不断地传输到目的地。接下来产生的问题就是何时应该停止在原始位置的执行，重新复制这些脏的页面，最后在迁移地点完成进程。中断进程执行和完成迁移是在特定的条件满足时实现的，决定条件的算法可以根据具体的脏页面与干净页面的比值，以及在一段固定时间标准内脏页面的不变数目或二者的结合。本方法有效地减少了停顿时间，但同停止和复制一样，仍在拷贝所有相关虚拟内存时浪费了时间和资源。并发复制也称为**预复制**，应用于 V-分布式系统中 [TLC85]。

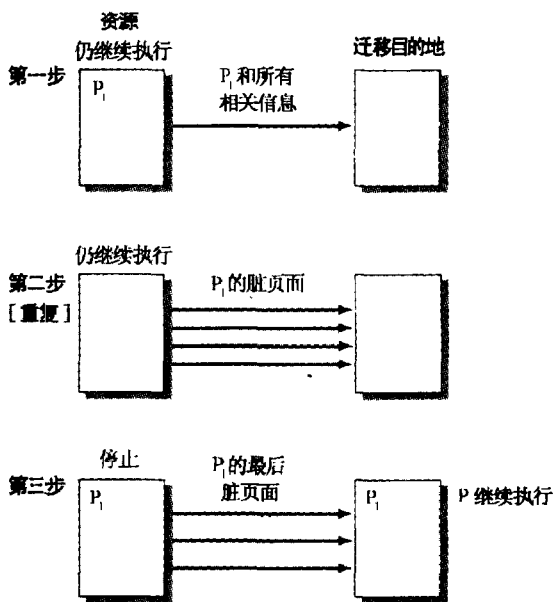


图 4-9 并发复制内存迁移

本方法有效地减少了停顿时间，但同停止和复制一样，仍在拷贝所有相关虚拟内存时浪费了时间和资源。并发复制也称为**预复制**，应用于 V-分布式系统中 [TLC85]。

98

第三个方法是引用时复制 (copy-on-reference)，当一个进程需要迁移时，它就停止。除了相关的虚拟内存外所有状态信息都转移到迁移目的地，接着进程继续在迁移地点执行，同时只从内存中转移进程执行时需要访问的那些页面。与被迁移的进程有关的页面可能存放在最初的位置或已转移到文件服务器。如果仍在原始位置，那么那里的资源仍在使用中，反之则被释放。如果利用服务器，所有的虚拟内存仍要在网络上转移，但不是所有的页面都必须迁移到目的地。这种转移在所有状态信息转移到迁移目的地之前发生，并允许原始位置上的内存资源立即释放。最后，利用服务器清除了原始位置与迁移目的地之间的依赖。图 4-10 描述了引用时复制方法。

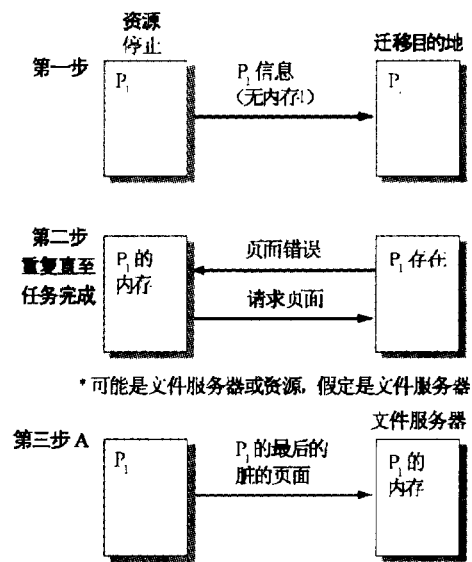


图 4-10 引用时复制内存迁移

4.6 小结

内存管理是操作系统中一个极其重要的部分，每一个进程的执行和所有数据的操作都必须驻留在内存中。表 4-1 给出了各种讨论过的能适用于高级系统的模型。实时系统可以使用给出的任何一种架构形式，所以没有包括在表中，然而内存管理为实时系统提高性能提供了机会。表 4-2 总结了本章介绍的提高系统性能的各种选择。

表 4-1 可用于高级系统的内存管理选择

内存管理模型	高级系统组件类型		
	并行 UMA	并行 NUMA	分布式的
简单内存模型	是		
共享内存模型		是	对小规模最好
分布式共享内存			是

表 4-2 内存管理的性能选择

方法/概念	性能选择
简单内存模型	运行好但需要避免颠簸
共享内存模型	通信和计算交叠 预读取数据 使用窃听式高速缓存
分布式共享内存	多读者/单写者：静态分布式算法 多读者/单写者：集中式算法 多读者/多写者（好的并发性） 允许暂时停顿 块大小 = 页面大小 硬件实现（至少是部分）
内存迁移	并发复制

有规则的分布式内存管理, 就像集中式系统一样, 仍将是一个活跃的研究领域。正如我们所讨论的, 大多数内存管理性能集中在有限的内存空间内执行和操作数据。因此有理由相信, 随着技术的进步和经济实力的提高, 内存的容量会增大, 内存管理问题就会变得不是如此关键。但同时由于系统需求的扩大和程序的日益庞大, 所以这种论断也不完全正确。要知道, 许多集中式系统曾经没有虚拟内存, 而只是拥有 16K 的 RAM!

4.7 参考文献

第 1 章提供的参考文献能提供更多的内存管理信息, 如 [PTM98]。与本章内容相关的一些传统的研究论文可参考 [AAO92, AGGL86, ARS89, BGGLOR94, ErCo92, ErCo93a, ErCo93b, Erick93b, Esk89, Esk95, Fu97, GaCo95, KeLi89, LaEl91, Lee97, LEK91, LiHu89, MoPu97, PTM96, SiZh90, SuZh96 和 TLC85]。

下面给出一些详细说明中讨论的两个系统在 Internet 上的资源起始地址。具体地说, CHORUS 主页为 <http://www.sun.com/chorusos/>, Amoeba 网页的地址为 <http://www.am.cs.vu.nl>。宾夕法尼亚大学分布式系统技术报告和研究实验室的网页 <http://www.cis.upenn.edu/~dsl/library.html> 包含一些关于在延迟方面的 DSM 的效率方面的研究论文。第 1 章的综合网页也为包含内存管理内容的具体文档提供了链接。

99

101

习题

- 4.1 就个人计算机的辅存, 回答以下问题。(提示: 查一下当前的商业杂志、制造商和销售商的站点。)
 - a. 现在有什么类型的硬盘驱动器?
 - b. 解释各种类型硬盘驱动器的区别。
 - c. 现在各种类型硬盘驱动器的价格分别是多少?
 - d. 现在各种类型硬盘驱动器的访问、寻找和传输速率是多少?
 - e. 现在硬盘驱动器的最大容量是多少? 为什么会有限制?
 - f. 现在只读 CD-ROM 访问、寻找和传输速率是多少?
 - g. 只读 CD-ROM 与可读写 CD-ROM 相比, 速率有什么不同? 为什么?
 - h. 现在 CD-ROM、可读写 CD-ROM 的价格是多少?
 - i. 现在 DVD 的价格是多少?
- 4.2 描述段相对于页面的一个优点。描述页面相对于段的优点。
- 4.3 使用更大尺寸页面的优缺点是什么? 使用更小尺寸页面的优缺点是什么?
- 4.4 写一个程序实现 FIFO 页面替换和 FIFO 第二次机会页面替换。程序应报告每一页准备放置的帧和页面错误的总数, 可以假设在开始时每个帧都是空的。(注释: 每次页面载入帧时发生一个页面错误。)使用 4 个内存帧和如下的页面访问序列测试程序。
 - a. 1, 2, 3, 4, 5, 2, 6, 2, 7
 - b. 1, 2, 3, 4, 5, 2, 1, 4, 2, 3
- 4.5 在共享内存模型中, 预读取数据有助于性能的优化, 讨论实现自动数据预读取的一个潜在问题。

- 4.6 描述和讨论以下每一个分布式共享内存访问类型的一个优点和一个缺点。
- 单读者/单写者。
 - 多读者/单写者。
- 4.7 编写一个程序实现具有集中式服务器的多读者/单写者 DSM。答案应包括一个集中式服务器程序和一个参与的客户机程序，服务器和客户之间利用进程间通信。另外，服务器和客户机应该把所有发生的行为和接收到的信息记录在文件中。使用 4 个参与客户和如下的请求序列及自己的测试数据测试程序。
- 客户 1 到客户 3 请求读页面 1。
 - 客户 4 请求写页面 1。
 - 客户 4 写页面 1。
 - 客户 2 到客户 4 请求读页面 2。
 - 客户 1 请求写页面 1。
 - 客户 1 请求写页面 2。
 - 客户 1 写页面 1 和页面 2。
- 4.8 讨论下面实现内存迁移方法的优缺点。
- 停止和复制。
 - 并发复制。
 - 引用时复制，数据保留在原始位置。
 - 引用时复制，数据在文件服务器。

第5章 并发控制

一旦有多线程控制或多进程访问共享数据和资源，就必须保证访问有秩序地执行，否则会产生错误的结果。本章将集中讨论并发控制问题。并发控制使进程在各自的跑道上运行，并保证在分布式或并行系统中顺利且正确地执行，因此它被认为是分布式计算中最重要的问题。因此，尽管它的部分内容在操作系统引论中多次出现，本章还是将详尽地对其进行介绍。5.1 节介绍互斥和临界区的概念以及三点测试方法。5.2 节到 5.6 节介绍各种解决互斥问题的方法。具体地说，5.2 节介绍信号量，5.3 节介绍管程，5.4 节介绍锁机制基础，5.5 节介绍软件锁控制，5.6 节介绍令牌传递方法。本章还包括互斥的可能结果的讨论，尤其是 5.7 节中关于死锁的问题。在本书第二部分的第 9 章使用事务管理的概念，扩展了数据的多访问问题，而第 10 章讨论与排序和同步相关的问题。

5.1 互斥和临界区

互斥（mutual exclusion）保证了共享资源的进程不同时使用同一个资源。如果分布式系统的用户到打印机前取文件时发现了他的文件与好几位同事的文件按页交叉地放着，肯定不是一件愉快的事情。用户希望他们的文件能够独占打印机，并连续地打印出来。使用互斥独占访问对于磁带机、打印机和文件来说是必要的。如果要保证峰值性能，一般不希望不必要地停止进程。访问共享资源的程序或代码称为**临界区**（critical region），只有这里需要禁止并发，称为并发控制，并通过信号量（5.2 节）、管程（5.3 节）或本章讨论的其他任何并发控制机制（5.2 节到 5.6 节）来实现。这些控制机制通过使进程停止和轮流访问资源的方式来确保互斥。图 5-1 显示了两个进程访问和改变变量 `foo` 的值及利用临界区的情况。在附录的外科手术调度程序中使用了一个基本的互斥操作。

105

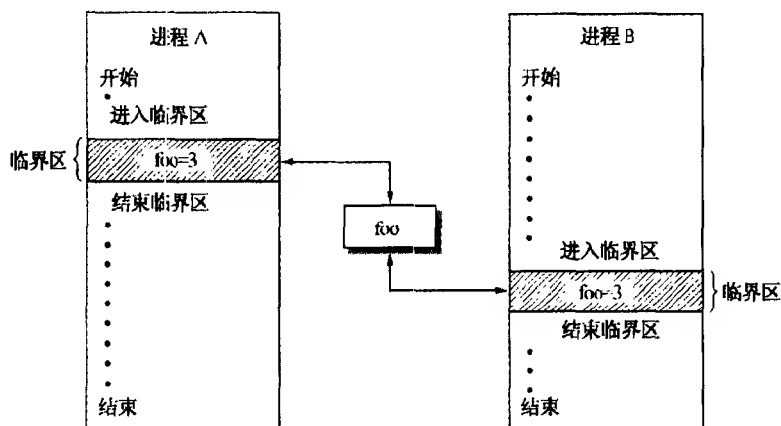


图 5-1 使用临界区保护共享变量

下面讨论本章给出的各种解决方案的评价基础。具体地说，互斥的解决方案必须通过如

下的三点测试：

第一点：解决方案能保证两个进程不同时进入它们的临界区吗？

第二点：解决方案能防止不试图进入它们临界区的进程产生的干扰吗？

第三点：解决方案能防止饥饿吗？（饥饿是由进程永久地等待进入临界区造成的，也称为无限等待。）

106

接下来讨论一些现有的解决互斥问题的方案。

5.2 信号量

使用信号量实现互斥的思想是 Dijkstra 在 1965 年提出的 [Dij65]，并由 Algol 68 编程语言实现。信号量是一个整型变量，它的值说明了受保护资源的状态。信号量只允许两个操作：raise 和 lower^①。当进程希望进入代码的临界区时，它必须执行 lower，离开时则必须执行 raise。raise 和 lower 检查信号量的值，并分别加 1 或是减 1。信号量检查和改变是作为原子操作完成的，也就是说，该操作是不可分的，并且不可间断。信号量的算法详见详细说明 5.1。详细说明 5.2 给出了进程使用信号量访问共享资源的范例。

详细说明 5.1

信号量算法

```

Let
    Semaphore SEM;
    // 初始化值是 0
    SEM=my_favorite_shared_device;

LOWER
    SEM=SEM-1;
    If SEM >=0
        Return(true);
    ELSE
        Wait (until_raise);
        Return(true);

RAISE
    SEM=SEM+1;    // 如果不只一个进程等待，系统选择
                  // 哪一个进程接受 raise
                  //

USING SEMAPHORE
    Bla Bla Bla; // 执行非临界区代码

    lower(SEM);  // 等待真值返回
    critical_region_code;
                  // 访问共享资源
    raise(SEM);
    Bla Bla Bla; // 执行非临界区代码

```

107

① P 和 V 源于荷兰语，是最初的操作的名称。

详细说明 5.2

信号量应用举例

假设进程 A、B、C、D 和 E 希望访问受信号量 SEM 保护的共享资源。这里通过跟踪 SEM 的值，介绍一个范例。我们还假设它们都使用详细说明 5.1 给出的算法，图 5-2 描述了所有的行为。

0. SEM = 0; //初始化值
1. SEM = -1; //进程 B 执行 lower, 进入临界区
2. SEM = -2; //进程 A 执行 lower, 等待 raise
3. SEM = -3; //进程 C 执行 lower, 等待 raise
4. SEM = -2; //进程 B 执行 raise, 离开临界区
5. //系统选择 raise 进程 C, 进程 C 进入临界区
6. SEM = -1; //进程 C 执行 raise, 离开临界区
7. //raise 进程 A, 进入临界区
8. SEM = -2 //进程 D 执行 lower, 等待 raise
9. SEM = -1 //进程 A 执行 raise, 离开临界区
10. //raise 进程 D, 进入临界区
11. SEM = 0; //进程 D 执行 raise, 离开临界区
12. SEM = -1; //进程 E 执行 lower, 进入临界区
13. SEM = 0; //进程 E 执行 raise, 离开临界区

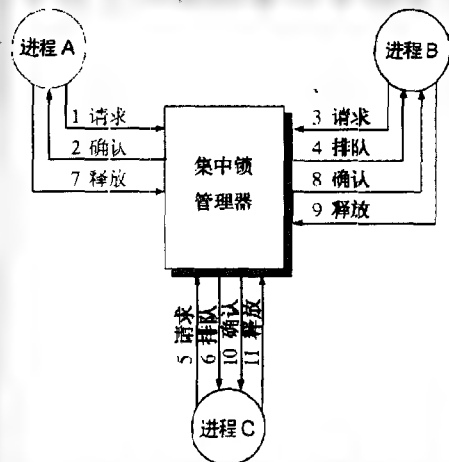


图 5-2 信号量应用范例

5.2.1 信号量的缺点

信号量的缺点包括它对程序员的依赖。信号量依靠程序员执行 lower 进入临界区并执行 raise 退出临界区。然而如果程序员意外地使用另一个 lower 离开临界区，这将会发生什么？（要知道为了如期完成任务，在凌晨 2 点编程，每个人都已经相当疲惫，以致易于发

生错误!) 信号量仍然处于被保护状态, 没有进程能够获得 raise, 与这些资源相关的系统则被死锁! 如果进程有一个不适时的 (即执行 lower 后但在执行 raise 操作前) 崩溃, 也会发生类似的死锁。有关死锁的问题在 5.5 节讨论。

5.2.2 信号量评估

我们根据三点测试对信号量进行评估。首先, 信号量能保证两个进程不同时进入它们的临界区吗? 能。其次, 信号量能防止不试图进入它们临界区的进程产生的干扰吗? 适当利用的情况下能。只有那些已经表示希望进入临界区的进程才使用信号量, 如果信号量使用不当 (如 5.2.1 节所指出的那样), 不再想进入临界区的进程就会阻碍等待进入临界区的进程。最后, 信号量能防止饥饿吗? 能。适当使用下, 信号量本质上不具有导致永久延迟的请求。然而, 正如我们所讨论的那样, 不适当的使用会导致死锁。总之, 适当使用信号量能够通过三点测试。详细说明 5.3 描述了 UNIX 系统 V 中对信号量的支持。最后应注意, 信号量在真正的分布式环境中很难实现, 并应尽量避免, 因为必须保持信号量数据的绝对一致性。关于保持一致性的内容参看第 9 章。

详细说明 5.3

UNIX 信号量支持

UNIX 系统 V 支持信号量作为进程间通信设施的一部分。在 C 或 C++ 编程中使用信号量, 必须应用如下的包含语句:

```
#include <sys/sem.h>
```

创建信号量的函数调用是 semget, 而 semop 执行各种信号量操作。所有细节的完全列表在本地系统 (输入 man semget, 或 man semop) 的手册 (在线 UNIX 手册) 中都有提供。另外, UNIX 系统中对信号量的硬件支持能通过关键字索引选项 (输入 man -k semaphore) 找到。

5.3 管程

管程是由 Hoare [Hoa74] 和 Brinch Hansen [Bri75] 提出的一种高层的同步原语。一个管程是一个编译器支持的编程语言结构。该编程语言结构允许一组过程、它们的变量和数据结构组合成一个包或一个模块。也就是说, 它是一个抽象数据类型。其他非管程进程可以调用管程内的过程, 但不能访问内部的数据结构。为了保证互斥, 管程不允许在任一给定时间, 在管程内有超过一个进程是活动的。如果一个管程内有一个活动的进程, 同时又有另外一个对同一个管程内过程的调用, 那么新的过程调用就会被阻塞, 直到活动的进程放弃控制。在管程内部, 编译器经常使用一个二进制的信号量。二进制信号量是只有两种值的信号量, 只有编译器而非程序员能使用这种信号量, 这样就避免了程序员误用这种信号量的可能。程序员只需要关心编译器的错误和内存崩溃 (内存崩溃是所有程序员的恶梦, 并且随着系统复杂度的增加而增加)。

5.3.1 条件变量

按照目前的定义, 管程包括若干过程和变量, 这就产生了如下的问题: 如果受到管程控

制的进程 A，为继续运行需要进程 B 执行一个特定的操作。假如 B 的操作不利用管程中的任何过程和变量，就没有任何问题。但是如果 B 需要利用管程内的过程来执行进程 A 继续运行所必需的操作，就会产生困境。要知道，在任何时间内，管程都不允许其内的进程超过一个。为了适应这种情况，管程引入**条件变量**。条件变量对管程是全局的，但又包含在管程内。类似信号量，条件变量有两个操作 wait 和 signal: wait 操作挂起调用进程，signal 操作继续执行进程。Wait 类似于信号量的 lower 操作，signal 却与信号量的 raise 不同。具体地说，如果进程没有被挂起，signal 操作则没有意义。

下面讨论如果进程 A 执行 signal，进程 B 等待允许在管程内激活，结果将会如何？下面列出两个选项及其相对优点：

- ◆ 进程 A 继续处于活动状态，直到离开管程。在它离开管程后进程 B 开始活动。这个方法的优点是没有上下文转换，因而效率高。潜在的问题是时间的花费和在进程 B 执行 wait 和接收 signal 之间发生了计算。这个方法是 Brinch Hansen 提出的 [Bri75]。
- ◆ 进程 A 挂起并允许进程 B 处于活动状态（上下文转换）。当进程 B 结束后，允许进程 A 重新处于活动状态（另一个上下文转换）。尽管这个方法涉及更多的上下文转换，但它保证了行为的准确。这个方法是 Hoare 提出的 [Hoa74]。

108

110

5.3.2 管程评估

我们根据三点测试对管程进行评估。首先，管程能保证两个进程不同时进入它们的临界区吗？能。其次，管程能防止不试图进入它们临界区的进程产生的干扰吗？能，管程在某一时刻只允许其内的一个进程处于活动状态，只有当前处于活动状态的进程或等待成为活动的进程在被激活的时候才能发生影响。最后，管程能防止饥饿吗？能，因为编译器实现了用来提供互斥的信号量。然而，在任何不使用共享内存的高级环境中，对管程来说却有一个严重的缺陷，就是编译器依靠共享内存实现信号量。总之，没有共享内存，就不能使用管程。详细说明 5.4 给出了 Java 中使用管程的概况。

详细说明 5.4

Java 的管程

Java 实现管程，并处理所有锁的设置和获取。程序员只要在对象和方法定义中利用 synchronized 说明什么资源需要并行控制即可。Java 中的条件变量有两个操作：wait () 和 notify () (signal)。条件变量的行为与前面介绍的 Hoare 的建议一致。如果有多个方法同时处于 wait () 睡眠状态，Java 基于先来先服务的机制唤醒对象。当线程同步时，同一时间只有一个线程执行，Java 还包含了 notifyall () 来唤醒所有的线程。

5.4 锁

正如我们所知道的那样，互斥涉及使用共享资源的进程，同时需要保证只有一个进程在使用上述共享资源。锁是日常生活中防止他人进入的一个工具，同样，它也能使其他进程不使用共享资源。一个锁有两种状态：上锁和未上锁。所以它能提供资源的并发控制和互斥。

111

如果一个进程希望使用共享资源，它首先检查这个共享资源是否上锁。如果资源没有上锁，这个进程就锁住资源继续执行。看起来很简单！然而考虑一下详细说明 5.5 的情况：两个资源都想进入临界区且同时发现资源未上锁，这导致一个共知的竞争状态，就可能会允许两个进程同时进入临界区！显然，这不是并行控制所期望的输出，也没有提供互斥。本节将要讨论避免这种竞争状态并成功实现锁的各种方法。

详细说明 5.5 竞争状态示例

假设进程 A 和进程 B 执行如下步骤：

1. 检查锁状态。
2. 发现锁处于未锁状态。
3. 加锁。
4. 执行临界区。

下面考虑的调度情况是两个进程竞争设置锁的竞争状态，结果导致不正确的执行。

1. 进程 A 检查锁状态。
2. 进程 A 发现锁处于未锁状态。
3. 进程 A 被调度程序送到等待队列中（因最大处理时间而阻塞）。
4. 进程 B 检查锁状态。
5. 进程 B 发现锁处于未锁状态。
6. 进程 B 加锁。
7. 进程 B 执行临界区程序。
8. 进程 B 被调度程序阻塞。
9. 进程 A 加锁。（记住，进程 A 在被阻塞前检查锁，而锁处于未锁状态！）
10. 进程 A 执行临界区程序。

142

5.4.1 轮转

轮转是大多数人在入学前就学会的一个很简单的概念。如果多个进程希望共享资源，那就让它们轮流享用吧。这个方法不允许进程放弃自己的机会。再假设进程 A 和进程 B 共享资源，如果进程 A 使用资源，那么接下来使用资源的必是进程 B，进程 B 也不能放弃。因为它不允许放弃，所以这种方法也称为严格轮转变更。轮转通过一个变量实现。如果两个进程共享资源，变量可以为 0 或 1。进程 A 初始化变量为 0，当它完成临界区的程序后，改变变量的值为 1。除非进程 B 进入临界区并从临界区退出，否则变量的值都不会也不能改变。如果进程 A 希望进入临界区，它首先检查变量的值，如果变量为 0，继续执行，否则等待。这种情况下，进程 A 会不断地检查变量的值。这种不断检查的过程称为忙等待。忙等待浪费系统资源，系统资源被不断地检查变量值而消耗掉。而不断的检查称为循环检查（spinning），不断检查的变量称为循环检查锁（spinlock）。也就是说，忙等待实现在循环检查锁上进行循环检查。

我们根据三点测试对轮转进行评估。首先，轮转能保证两个进程不同时进入它们的临界区吗？能，进程只能根据轮流状态进入临界区，进程不能同时改变变量。其次，轮转能防止不试图进入它们临界区的进程产生的干扰吗？不能，如果进程 B 不想进入临界区，它会阻碍进程 A 进入临界区。最后，轮转不能防止饥饿。考虑进程 A 不试图进入临界区改变变量的值，进程 B 将永远禁止进入临界区，因而会遭受饥饿。总之，这个解决方案不能通过三点测试，但如果并入第 2 章介绍的轮询方法就会纠正轮转的缺陷。作为练习，评估一下修改之后的方法。

5.4.2 原子操作和硬件支持

利用锁来解决互斥问题的系列方案还涉及原子操作。第一种解决方案通过实现原子的（不可分的）检查和设置（check-and-set）操作，防止了如详细说明 5.5 中介绍的竞争状态。因操作是原子的，另一进程不可能执行任何与检查和设置锁操作交织的操作，这个操作也称为测试和加锁，通常由硬件支持以保证原子操作。如果两个进程同时想检查和设置，系统通过使用硬件支持决定哪一个进程先执行。

113

涉及原子操作的第二个解决方案是交换。交换一般使用一个临时变量，执行三条语句。原子交换能够用一个表示共享资源处于被锁状态的值交换出锁状态的当前值。这时进程测试它交换出来的变量值。如果交换出来的值说明已经被锁，进程就不能继续执行，变量仍处于被锁状态。但如果交换出来的值说明未锁，因变量已处于被锁状态（由交换导致），进程就会进入临界区，如图 5-3 所示。同样，如果两个进程同时想执行交换，系统会通过使用硬件支持决定哪一个进程先执行。

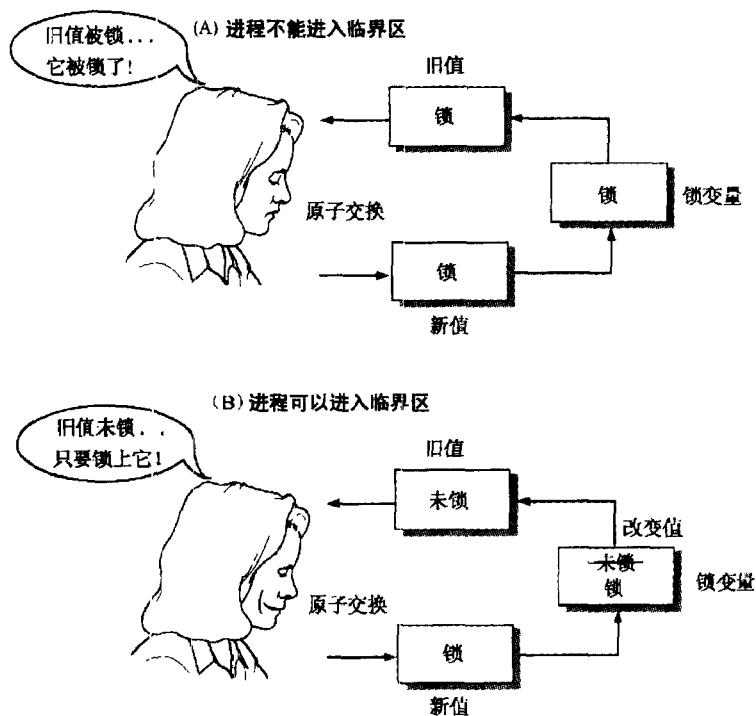


图 5-3 原子交换

我们根据三点测试对原子操作方案进行评估。首先，原子操作方案能保证互斥。其次，它能防止不试图进入它们临界区的进程产生的干扰。第三，希望访问受原子操作保护的共享资源的进程不会饥饿，而总会被安排并获得进入临界区的机会。对高级系统评估这种方案很困难。原子操作很难在并行计算机的硬件中实现，但却在一些系统中使用，多物理计算机的分布式系统需依靠软件实现。

5.5 软件锁控制

当硬件支持锁不可行的时候，需要使用软件实现。分布式解决方案经常利用软件提供各种不同类型的锁，从而与第4章讨论的内存管理调度（具体内容是提供了读锁和写锁）一致。分布式锁控制的解决方案利用了集中式锁管理器或分布式锁管理器。下面就讨论这两种基本的方法。

5.5.1 集中式锁管理器

与使用集中式控制的分布式计算中的每一个解决方案一样，利用集中锁管理器导致了单个关键元素和单点失败，所有这些方法在分布式实时应用中不可行。如果锁管理器崩溃，整个系统都会受害。在集中式管理点周围会引起流量的增加，如果需求足够大还会导致系统瓶颈。锁管理器执行的功能大多数与集中式系统的硬件所执行的功能一样，服务器维护进程请求进入临界区的信息以及允许进程访问的资源的信息。集中锁管理器方案涉及三种必需消息和一种可选消息：

- ◆ Request。这个消息从请求的进程传输到集中式锁管理器，它的目的是请求进入临界区。
- ◆ Queued。这个消息是可选的，由集中式锁管理器传输到请求的进程。在请求不能立即应答的情况下使用此消息，用于通知进程已收到请求，请等待处理后的消息。不能立即应答的请求按照先来先服务的次序放置在队列中。如果没有使用这个可选的消息，成员就很难区分一个死的集中式锁管理器和一个忙的集中式锁管理器。
- ◆ Granted。这个消息从集中式锁管理器传输到请求的进程，只在进程可以进入临界区的时候传输。如果请求被接收但不能接受服务，请求的进程收到可选的“Queued”消息或什么都收不到。
- ◆ Release。这个消息从已经接受过服务的请求进程传输到集中式锁管理器，它说明进程已经结束了它的临界区程序，并释放锁。这时候，集中式锁管理器发送“Granted”消息给队列中的下一个进程。

图5-4描述了一个带有三个都请求进入临界区进程的集中式锁管理器的功能，例中使用了三个必需消息和可选的“Queued”消息。

我们根据三点测试对集中式锁管理器进行评估。首先，它能保证两个进程不同时进入它们的临界区吗？能，集中锁管理器在某

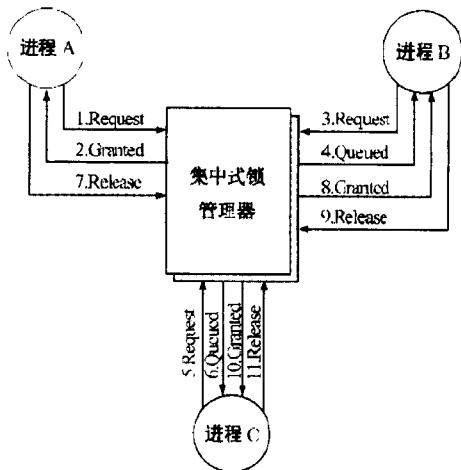


图 5-4 集中式锁管理器

一时刻只允许一个进程进入临界区，所以成功地保证了互斥。其次，它能防止不试图进入它们临界区的进程产生的干扰吗？能，当某一进程进入临界区时，只有一个执行临界区程序的进程或队列中的其他进程能够产生影响。最后，它能防止饥饿吗？能，这个算法按照接收请求的顺序提供服务，因而很公平，能够在某一时刻所有提交请求的进程提供服务。

5.5.2 分布式锁管理器

实现锁管理器的分布式解决方案有两种必需消息和一种可选消息，虽然它与集中锁管理器有许多相似之处，但还是有一些区别如下：

116

- ◆ 谁接收这些消息。
- ◆ 谁发送这些消息。
- ◆ 进入临界区的必要条件是什么。

下面讨论这些消息，包括关于发送者和接收者的信息。

1. Request。这个消息从请求的进程传输到系统中所有的成员。它可以分别传送给每个成员或利用组通信机制，目的是请求进入临界区。所有的请求包括一个源时间戳，第10章给出了分布式环境中利用时间戳的算法。

2. Queued。这个消息是可选的，由接收到“Request”消息的进程传输给请求的进程。它可以用于通知进程请求已经收到，请等待下一步处理后的消息。在请求不能立即答应的所有情况下都可使用此消息。成员在执行临界区程序的时候或之前已经发出请求消息（由时间戳决定）但仍未执行的情况下不会答应请求。任何被拒绝的请求都会放置在队列中，直到临界区程序结束或请求完成。如果没有使用这个可选的消息，就很难区分一个死的成员和一个忙的成员。

3. Granted。这个消息在两种情况下从所有的成员传输到请求的进程。第一，如果成员不在临界区且没有一个更早时间戳的进入临界区的请求，就发送一个“Granted”消息。第二，当成员有一个排队的请求，它就会在自己完成临界区程序后发送一个“Granted”消息给队列中其他所有的成员。这种情况基本等价于集中式方法的“Release”消息。如果接收的“Request”消息不能被答应（如进程在临界区或有一个更早时间戳的请求），请求的进程就收到可选的“Queued”消息或什么都收不到。

下面讨论许可的问题。在集中式方法中有一个锁管理器，“Granted”消息是进入临界区的绝对权威。在分布式锁管理器算法中，“Granted”是应答许可的一张选票，而不是绝对的许可。利用分布式锁管理器的早期解决方案需要意见一致的决定：成员必须收到其他进入临界区的每一个成员的“Granted”选票。这样很复杂，很难知道谁参加了分布式系统和在某一时刻有多少成员，因为成员进入系统并因休息、崩溃和网络故障而离开。如果必须要一致的决定，所有成员都是关键因素，这是不易接受的！一个变通的办法是需要一个多数选票：某个成员接收到大多数成员的“Granted”消息就能进入临界区。

117

我们根据三点测试对分布式锁管理器解决方案进行评估。首先，它能保证两个进程不同时进入它们的临界区吗？能，使用意见一致的决定，所有成员一起决定允许某一进程继续执行；使用多数选票，大多数成员决定允许谁进入临界区继续执行。这里只有一个多数，因多个的多数需要重复的成员，且需要成员在发送“Granted”消息之前检查更早的请求。因此，多个进程不能同时接收到进入临界区的适当许可。其次，它能防止不试图进入它们临界区的

进程产生的干扰吗？不能和能。在意见一致决定实现中，进程如果没有其他所有成员的“Granted”选票就不能继续；如果某一个成员崩溃，就不会再有其他的成员继续运行，这样崩溃的成员就阻碍了其他进程进入临界区。所以，意见一致实现在这点上不能。相反，因大多数选票的方法允许单个系统失败，所以能通过第二点的测试。最后，它能防止饥饿吗？因请求按照排序的方式被服务，所以所有的请求都能被服务，没有进程会遭受饥饿。总之，意见一致的方法不能通过三点测试，多数选票的方法能通过。

5.6 令牌传递互斥

互斥的令牌传递解决方案目的是在通过网络相联的多处理器（并行或分布式）中使用，它由一个令牌和一个遍历所有处理器的路径——逻辑环组成。为了进入临界区，进程必须获得这个令牌。因为只有一个令牌，所以只有一个进程可以进入临界区。当进程退出临界区时，令牌释放到系统中。如果没有进程希望进入临界区，令牌就持续不断地循环。虽然这个方法很理想，但也会引起一些问题。

- ◆ 如何判定令牌是丢失还是长时间地使用？
- ◆ 如果拥有令牌的进程崩溃了很长时间将如何？
- ◆ 如果一个处理器离开（自愿或因故障）系统，如何维护逻辑环？
- ◆ 如何辨认和增加加入逻辑环的处理器？

这里的某些顾虑能通过指派一个或两个特定区域作为监控器来解决。监控器会偶尔发送消息询问令牌持有者，如果没有应答，则在网络中放置一个新的令牌。监控器登记新成员同时记录丢失的成员。确认丢失成员最简单的方法是通过它的邻居。当处理器试图传递令牌的时候，它等待确认。如果没有确认，则再传一次；如果再次没有确认，就认为它的邻居已丢失，并通知监控器记录和指派新的邻居。

我们根据三点测试对这个解决方案进行评估。首先，它能保证两个进程不同时进入它们的临界区吗？能，逻辑环在任何时候都只有一个令牌，即使利用两个监控器，令牌丢失的时候也只会系统中放置一个新的令牌。其次，它能防止不试图进入它们临界区的进程产生的干扰吗？能，一个不相关的进程只会传递令牌，不会阻碍。最后，它能防止饥饿吗？能，所有的进程都受到公平和平等待遇。

5.7 死锁

到目前为止，本章都在讨论互斥，本节将要讨论的是互斥可能产生的一个副作用。具体地说，互斥是四个导致死锁的分配条件之一。如果以下四个条件成立，就会发生死锁吗？

- ◆ 互斥：就像我们所知道的那样，它是提供进程独占式访问资源的一种资源分配方法，用于防止资源的并发共享。
- ◆ 非抢占资源分配：一种系统提供进程的资源访问，但系统不能强迫进程放弃资源的控制。
- ◆ 持有和等待：进程持有一个资源并且等待另一个资源。
- ◆ 循环等待：在资源分配图中有一个循环路径，其中的每个进程持有至少一个资源，而且这个资源在进程获得该环路中另一个成员所持有的资源之前不会被放弃。如果没有持有和等待，就不会发生循环等待。

因此，死锁是具有不适当资源分配的系统状态，导致两个或更多的进程不能改变状态以完成运行。如果不存在上述的任一条件，就不会发生死锁。

让我们看一下图 5-5 (A) 资源分配图中描述的两个进程和两个资源之间的一个简单的死锁例子。所有方向朝资源的箭头是请求资源，而离开资源的箭头代表分配。每个进程已有一个资源并想得到另外一个资源。除非它得到并使用了其他资源，否则不放弃已有的资源 (循环持有和等待)。注意如果引入抢占，进程就会被迫放弃资源，那么循环也就断开了，因此也不会有死锁，如图 5-5 (B)。如果进程能并发共享资源 (没有互斥)，那么循环也不再是问题，同样不会有死锁，如图 5-5 (C)。最后看一个没有循环的例子，如图 5-5 (D)，没有循环，进程就不会被禁止运行，因而就没有死锁。死锁只在进程独占资源，但不能抢占并涉及一个分配循环的时候发生。

120

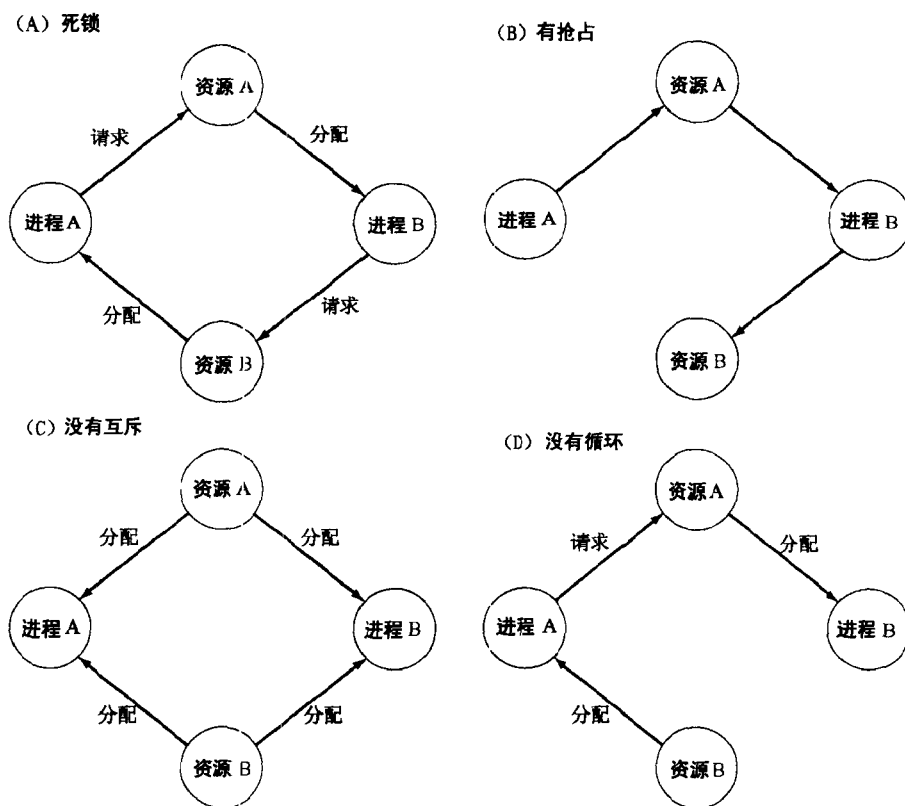


图 5-5 资源分配图

如果资源是分布式系统的缓冲空间，那么死锁一般称为**通信死锁**。如果通信死锁只涉及两个位置，那么它就称为**直接存储转发死锁**；如果涉及多个位置，则称为**间接存储转发死锁**。

如何处理死锁呢？使用助记符 PAID 记住相应的方法：防止 (Prevent)，避免 (Avoid)，忽略 (Ignore) 或检测 (Detect)。接下来讨论每一种方法。

5.7.1 防止死锁

为了防止死锁，必须防止前面提到的任何一个分配条件。本章一直讨论第一个分配条件

——互斥，它是访问如打印机资源的必要条件，所以不是避免死锁的一个选择。打印机也体现系统为什么不采取抢占——第二个分配条件，因为这样会产生多个打印任务相互干扰，可能会打印到同一页上。如果没有持有和等待，循环等待也不会发生，所以只剩下一个分配条件可以打破。打破这个分配条件的方法有好几个，但都有缺点，下面分别介绍。

- ◆ 只允许持有单一资源。如果一个进程只允许持有一个资源，而不允许持有多个资源，就不会有循环！这个方法很麻烦，因为有些进程需要同时访问多个资源。
- ◆ 预分配资源。这个方法强迫每个进程在初始化执行之前获得需要访问的所有资源，所以资源有效利用的问题就出现了。假设有个进程需要三周完成运行，在运行结束时需要访问资源，预分配会迫使进程专用这个资源并在使用它之前的三周时间内一直让它处于无用状态！这样效率就会很低。
- ◆ 强迫释放再请求。这个方法强迫每个进程在另外请求资源之前释放所有已分配的资源，这样允许任何等待被其他进程持有的资源的进程能够继续运行，因而防止了循环持有和等待，但对有些应用这个方法就不太实用。
- ◆ 按序获得。这个方法为所有资源编号，进程只能请求序号高于自己所拥有的资源的资源。因为不能请求更低序号的资源，所以不可能产生循环，也就避免了死锁。但在分布式环境中用这种方式对所有的资源进行排序，工作繁重并且效率很低。尤其是谁都不愿意被分配到序号最高的资源。
- ◆ 资历规则。这个方法使用时间戳（第 10 章讨论）保证进程上时间戳的惟一。如果旧进程（根据时间戳）请求一个资源，那么任何持有这个资源的进程都必须放弃它。资格老的进程总会赢，而年轻的进程必须小心地保持自己的完整性以防死锁（详见第 9 章的事务管理）。

121

5.7.2 避免死锁

回忆集中式系统中的死锁避免算法，所有的算法都强迫进程提前给出需求。在需求给出以后，避免死锁的算法（因 NP 完全性的原因而特别缓慢）决定能否进行安全分配。如果能，允许进程继续运行并分配给所需的资源。但同时也会出现问题，首先没有进程能够准确预计需求，其次算法太慢以至不能用于集中式环境中。因此，很少有人愿意增加网络延迟以换取现成的慢算法。在分布式环境中使用这个算法将会更慢，所以下面接着讨论忽略死锁。

5.7.3 忽略死锁

忽略死锁是 UNIX 采用的方法，并受到多数人欢迎，它迎合了哲学观点“别担心，会好的”，其实这个方法只是选择忽略问题。如果系统管理员发现系统有问题，或者从用户那里听到抱怨，那么管理员进行调查（或派别人调查）并试图解决问题。通常，死锁进程的主人（只认为慢）开始烦躁，这位烦躁的用户一般取消进程，从而打破循环持有和等待，问题也迎刃而解了！这个方法也称为恢复。

5.7.4 检测死锁

在所有处理死锁的 PAID 方案中，检测是研究最多的领域。（多少人会同意研究忽略？）检测方法不关心死锁的产生，只注重寻找已经产生的死锁。一旦找到死锁，就会采取相应的

122

方法消除死锁。像我们以前所讨论的问题一样，检测死锁的方法在集中式和分布式系统中各有优缺点。在这个领域内有两篇经常被引用的调查：[Kna87] 和 [Sin89]。这里将集中讨论一个异乎寻常且成功的解决方案，这就是 Chandy, Misra 和 Haas [CMH832] 提出的分布式解决方案。这个方案基于探测消息，就是不论进程获取资源失败或时间用完都会发送具体的消息。探测消息包括如下三个字段：

1. 阻塞进程的标识。
2. 发送这个消息的进程的标识。
3. 消息发送的目标进程的标识。

发送探测的过程如下：

1. 阻塞进程 (A) 初始化探测消息。探测消息被发送给等待的资源持有者 (B)，始发者 (A) 把自己的标识放在探测消息的字段 1 和字段 2，把资源持有者 (B) 放在字段 3。
2. 资源持有者接收探测消息。如果位置 (B) 没有等待另外位置的资源，它就终止探测消息。但如果位置 (B) 在等待另一位置 (C) 的资源，它就根据步骤 3 转发探测消息。
3. 转发探测消息给 (C) (所有位置)。探测消息做如下改动：字段 1 不变，字段 2 成为 (B) 的进程标识，字段 3 标识新的接收者 (C)。
4. 如果字段 1 = 字段 3，则存在死锁；否则从第 2 步开始重复。

如果字段 1 等于字段 3，则出现死锁。也就是说，探测消息又被送回它的始发者。举一个简单的例子，考虑直接存储转发死锁的情形，使用 (A, A, B) 发送第一个探测消息，位置 B 接收到消息后使用 (A, B, A) 把探测消息发送给 A，这样循环就被识别，从而发现了死锁。

既然死锁已被识别，我们有三种选择来消除死锁。第一，可以谋求系统操作员的帮助。第二，可以终止进程（就像忽略方法！）。第三，回滚系统，这是第 9 章讨论的事务的一个特征。假如你是系统操作员，你选择终止一个或两个进程即可继续。最明显的选择是优先级低、且没有使用大部分系统资源的进程，同时也不是你老板的。为了防止将来可能的饥饿，终止了的进程通常会增加一个优先级，这样不至于在将来发生死锁的时候再次成为受害者。

123

5.8 小结

当多进程或多线程控制计算时，并行控制是必要的。没有并行控制，应用就不能完全利用分布式或并行环境中的资源。本章已经讨论了并行控制得益于硬件、操作系统和应用开发语言之间的协作，表 5-1 对其做了总结。最流行的现代语言提供了支持互斥和访问临界区的一些机制，给出的方法通常能用于任何多线程应用。附录中的外科手术调度程序提供了一个使用并发控制的多线程应用的范例。

表 5-1 硬件、系统和语言对并发支持的总结

支持程度	提供的工具
硬件	原子操作
	循环检查锁
	互斥体硬件操作
	高速缓存并发控制
系统支持	使用硬件支持的系统库

(续)

支持程度	提供的工具
语言支持	系统支持多线程应用的并发
	锁管理器
	令牌传递互斥
	信号量
	临界区
	管程
	使用系统库进行互斥体操作

本章还讨论了在访问共享数据中允许互斥的一个潜在问题——死锁。在所有 PAID 方法中，检测最受欢迎且最可行。死锁检测通常很难，资源越多，则问题越复杂。经常的网络延迟令“挂起”和“慢速”很难区分，也许这就是忽略算法的魅力！如果不忽略分布式环境中的问题，检测的探测方法也许就是目前为止最流行和最优雅的方法了。

124

5.9 参考文献

关于并发控制的综合资料可在许多介绍操作系统的书中找到。利用并发控制编程的资料包括 [Cha97, Cou98, DeDe94, DeDe98 和 RoRo96]。与本章内容相关的另外一些传统的研究论文包括 [AgAb91, Bri75, CaRo83, CMH83, Dij65, Fu97, Hoa74, Joh95, JoHa97, Kna87, RiAg81, Sin97 和 Tho79]。

SUN Microsystems 网站包括了使用并发线程控制的编程范例，这些例子可以用作模板，网址是 <http://www.sun.com/workshop/sig/threads/Berg-Lewis/examples.html>。另外，第 1 章中给出的网址也是寻找并行控制内容的最佳网址。

习题

- 5.1 互斥对于保护共享数据是必需的。列出三种需要互斥的数据访问应用。
- 5.2 数据互斥能够用于不同的粒度层次。需要保护的数据单位可以是单个的数据、单个的记录或一个记录集。讨论不同的粒度层次的优点和缺点，并给出实例。
- 5.3 退出一个临界区时意外的信号量 lower 操作是常见的编程错误。请描述一个对信号量 raise 和 lower 操作的可能的改变方法，该方法可以避免该问题。
- 5.4 假设有一个应用允许对进程分配优先级。这些优先级将用于决定在多个进程处于阻塞状态时哪个进程接收一个信号量的 raise 操作。这种方案能通过三点测试吗？请解释你的答案和必需的假设条件。
- 5.5 请描述一个替代忙等待的方法，并分析此方法的优缺点。
- 5.6 在第 2 章中我们学习了用轮询来确定调度。考虑一个使用锁互斥的方案。该方案中使用一个轮询其守护资源的用户的连续循环。当一个进程被轮询，它可能接受锁定该资源的机会，或者放弃直到下次被轮询。请根据三点测试要求简要评价一下这种方案。同时描述这种方案的长处和短处。
- 5.7 请问在集中式的锁管理系统中如果不使用“Queued”的消息，会引起什么潜在的问题？

125

- 5.8 当使用一个集中式的锁管理器时，如果一个进程被授权访问它的临界区后死亡，会发生什么样的问题？并请提出避免此问题的方案。
- 5.9 课程注册是所有学生都经历过的事情。对于某门指定的课程，只有有限数目的空位。每门课注册的学生数量都不允许超过所拥有的空位数目。另外，如果只允许一个进程访问注册学生所需要的数据，那么注册过程将顺利进行。因此，和每门课程有关的数据必须通过并发控制来保护。请编写一个程序来实现课程编排。你必须考虑该程序的多个副本能运行，并且这些副本访问相同的数据。你也可以假定你被世界上最小的大学雇用，该大学只有一门课程。彻底地测试你的程序，选择以下方法中的一种实现。
- 选项 A：用信号量实现你的方案。
- 选项 B：用管程实现你的方案。
- 选项 C：用锁实现你的方案。
- 5.10 自动柜员机是日常生活中的一部分。请用一个模拟程序来实现一个简单的自动柜员机。确保使用一种并发控制机制来保持账户平衡，因为可能有多人拥有对一个指定账号的存取卡。你的方案必须着重考虑并发控制问题，你可以选用以下一种方法来实现。
- 选项 A：用信号量实现你的方案。
- 选项 B：用管程实现你的方案。
- 选项 C：用锁实现你的方案。

第 6 章 基于对象的操作系统

基于对象技术已经出现在包括操作系统在内的计算机科学的各个方面。在这一章里，我们将介绍对象及其相关的术语（6.1 节）。通过详细讨论基于对象的操作系统 Clouds 来分析对象如何运用到操作系统中，特别是 6.2 节展示了 Clouds 是怎样实现对象、线程，以及它对存储系统全面的设计。在 6.3 节我们将讨论 Chorus 的面向对象子系统 COOL 的体系结构。6.4 节介绍 Amoeba，这主要集中于它在标识、保护和通信方面对对象的处理。两个主要的对象模型在 6.5 节和 6.6 节讨论。6.5 节特别详细介绍分布式组件对象模型（DCOM），这个模型最初由微软提出的，但目前由一个独立的机构管理，并对所有的 UNIX 和 Windows 平台是可用的。6.6 节介绍一个流行的基于对象的标准——称为公共对象请求代理程序体系结构（CORBA）。

127

6.1 对象介绍

操作系统是一个程序，通常由一般的程序语言如 C 和 C++ 来编写。虽然对象出现在计算机科学的每一个方面，但是对象及其相关术语还没有一个大家都认可的、严格的、一致的定义。本节我们重点讨论已广泛接受的定义。6.1.1 节介绍对象的常见定义，6.1.2 节对基于对象系统的优点进行讨论。

6.1.1 对象定义

对象是一个如图 6-1 描述的封装所有相关服务和数据的抽象。将对象服务定义为函数，在基于对象系统中函数也称为方法或成员函数。在本章中我们使用术语**方法**（method）。方法不能被独立地访问，只能通过它们的对象来访问，若没有对象方法就毫无意义。以 walking 为例，walking 意味着什么？如果我们对此作进一步的描述，如一个人 walking 或一条狗 walking，那么 walking 就有了某种意义。方法只有依附于它的拥有者——对象才有意义。当客户方对对象发出**请求**（request）时方法就被访问了，这样对象服务请求来自于客户方。请求能通过参量接受信息并且可以返回结果。被一个特定对象执行的方法的完整集合定义了该对象的行为。例如，一个人的行为是所有的“方法”如走、讲、唱、低语、呼喊、思考、吃、喝、睡、跑、坐、跳等的集合。

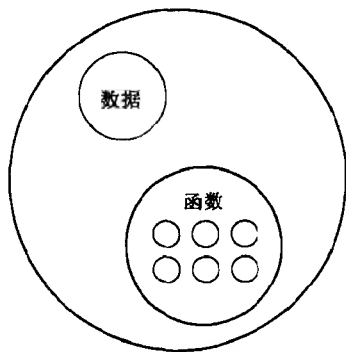


图 6-1 一个对象

在基于对象系统中通常允许方法有多种定义，只要调用中的参数与定义了的方法匹配，允许出现实际的行为由请求中的参数来决定的通用请求。当允许有多个方法例示时，方法例示常称为**构造器**（constructor）。同时允许方法的多个定义通常称为**函数重载**（function overloading），在第 3 章详细说明 3.4 中有相关的例子，在那个 Java 语言的例子中对于套接字对象

就有多个含有某些默认参数值的构造函数。

基于对象系统常常允许一个对象的多个实例。这种特点使得对象可以共享一个共同的实现。例如,如果我们有一个 person 对象,我们可以有两个该对象的独立实例,如 Steven 和 Sue。一个对象的抽象定义通常称为类(class)。当使用这个术语时,它的实例就称为对象。程序类的实例在 Java 和 CORBA 中经常被称为服务工具。

此外,对象可以通过继承(inheritance)来部分地共享实现细节。继承允许对象用现有的对象来定义,当允许继承时,基于对象的系统称为面向对象系统。某些实现允许对象可从多个对象继承特性,这种类型的继承称为多重继承。

128

6.1.2 对象的评价

对象有许多优点。使用对象最大的优点之一是数据抽象和封装。这种封装促进程序的模块化,并使之有清晰的接口。此外,对象使得程序有很强作用域及类型检查,通过禁止不恰当的数据操作和对象动作相应地增强了安全性。在基于对象系统中,分布式的迁移单位和在共享存储器中使用的单位是对象。在面向对象系统中继承特性使得可以进行增量设计和开发。详细说明 6.1 介绍了一个纯对象模型的变型——面向主题编程(SOP)。SOP 在大型软件系统(包括分布式操作系统和应用)中有着良好的前景。

详细说明 6.1

面向主题编程(SOP)

面向主题编程曾在许多论文中论述过,这其中包括 [HaOs93, OsHa95], 并在 [MDCM99] 中就其在分布式环境中的潜在的应用进行了点评。IBM 公司 T. J. Watson 研究中心的一个研究小组为了支持大型软件项目的开发而设计了 SOP, 允许开发小组的分组使用传统的面向对象技术来编程。传统的面向对象开发模式与 SOP 的区别是当这些开发小组将程序合并在一起组建大的软件时, 每个开发小组对这个大系统有着自己的视图。在主题视图层次中, 定义和实现特定视图的类集合(或类集合的一部分)就形成了主题。组合在一起的主题集合连同合成的组合主题就形成了域。通过使用不同的域的概念, 一个已知的类可以在不同的域中使用多次。类的行为依赖于它是怎样组合成域的方式(依据称作组合规则的规则)而定。实际上, 每个域的结构被封装起来, 这样一旦所有的域开始组合在一起时每个域就和其他的域隔离。

形式上, 主题代表了构成问题域的多个类的主观视图。可以将面向对象编程看作限制用户为一个主题视图, 这样增加了开发人员对全局知识的依赖或可能带来不必要的副作用。SOP 允许对同一对象有多个主题视图。由于这些视图对象可以有不同的接口, 开发人员只需知道和理解他所工作的域即可。面向主题模式要求考虑应用所需要的对象的接口, 不同的应用可能需要不同接口。我们不用考虑其他的应用, 因为它们存在于我们的空间和时间之外。使用 SOP 仍然必须确定对象但也只限于本领域中的对象, 这些对象无需知道系统中的所有对象(即那些主题域以外的对象)。此外只需掌握对象的上下文环境中所支持的消息即可。这种行为由对象来提供, 行为可以由应用程序开发人员添加到对象中。因此, 应用程序开发人员与“类主人”之间的通信就大大减少了 [OsHa95]。

这种系统类型的意义是每个本地域不受全局域改变的影响。这样该类型的模型能增强系统的位置、迁移及并发透明性。虽然该方法还处在研究阶段而且不是定位于操作系统，但是将该模型应用于大型系统如分布式操作系统将会是很有前途的。在分布式环境中，与面向对象编程相比 SOP 有许多优点，具体见 [MDGM99]。此外类似的技术在语言支持环境中也曾讨论过 [LoKi97]。

虽然使用对象的优点得到了认可，但是面向对象技术在高级环境中还存在一些争议。使用对象有缺点吗？是的，尽管大多数与面向对象系统而不是与基于对象系统有关。对象继承特别是多重继承难于管理。任何对父对象所做的改变将影响从它派生出来的子对象，而且对带有多重继承的大型系统的设计和管理难于表示和编写文档。此外，对象的行为依赖于派生它的父对象的行为。为了理解对象的行为通常必须理解父对象的行为，这意味着要对整个系统有着完整的认识（这是在大型分布式系统中不愿意看到的）。但是操作系统常常由受控的一组人来开发，而且基于对象操作系统并不是要求在系统上开发的应用程序均应采用基于对象方法。基于对象的系统和语言一般要慢一些，在有网络延迟的分布式环境中或在有严格时间限制的实时系统中，采用已知的较慢的技术通常不令人满意。尽管有以上缺点，但也存在一些成功设计的基于对象和面向对象的操作系统。

6.2 Clouds 对象方法

Clouds 是一个基于对象的分布式操作系统，它由佐治亚理工学院开发，在 [DLAR91] 中有相关的讨论。Clouds 是一个带有本机内核（即内核不运行于其他操作系统之上）的通用操作系统。内核称为 Ra^{\ominus} ，是一个分层的内核而不是微内核。Clouds 可应用于集中式或分布式应用中，而且这些应用可以由面向对象或非面向对象语言编写。6.2.1 节将讨论 Clouds 的对象，在 6.2.2 节中介绍 Clouds 的线程，最后在 6.2.3 节中对 Clouds 的存储系统进行讨论。

6.2.1 Clouds 的对象

Clouds 基于流行的面向对象的编程模型及对象线程模型。在本节中我们将介绍 Clouds 的对象。每个 Clouds 对象由数据和方法组成。每个完整的 Clouds 对象是持久的并位于一个虚拟地址空间中。并且其对象不允许多重继承。对象通过消息来进行通信，相应地引起一个方法的执行。这种方法可能会改变数据的值或使另外的对象接收到消息。当调用的方法执行完毕，通过返回参数值使得对象能对发送者的消息做出回应。消息由线程的执行而产生，这也就是模型取名的原因。这些消息通过调用带有输入参数的入口点来传给对象，参数严格限于使用数据而非数据的地址。所有地址都没有多少意义，因为它们对于一个已知对象来说是相对性的。一个用户能通过指定对象、入口点及 Clouds 的 Shell 参数来调用 Clouds 的对象。

Clouds 的对象被认为是“重量级的”，它最适合于大粒度的程序。Clouds 对象又是被动的，因为它不含有进程或线程。具体地说，一个 Clouds 对象由以下项组成：

- ◆ 用户定义的代码。
- ◆ 持久数据。

⊖ Ra 是埃及的太阳神。

- ◆ 用于临时内存分配的易失堆。
- ◆ 用于分配内存的持久堆。

在物理上所有的对象都存放于数据服务器中，但是 Clouds 提供了位置透明性，这样所有数据都可以从任何系统服务器中访问。由于每个对象都有一个惟一的全局性系统级名字，从而使透明性得到了进一步的增强。一个用户不用对对象的系统名字负责，他可以赋予对象任何名字而由名字服务将该名字翻译成惟一的系统名字。

131

Clouds 通过让对象支持存储和线程的抽象来维护计算和存储的正交性，以解除这两者的关联。Clouds 的对象线程模型的实现使得可以对 I/O、进程间通信、信息共享、长期存储以及原子的和可靠的计算进行统一的处理。

6.2.2 Clouds 的线程

线程在对象之间迁移并从每个对象的入口点开始执行。当线程迁移时，它们执行对象内的代码，这样各种线程代表了所有用户在每个线程中的活动，它表示了程序执行的逻辑路径。一个线程可以由两种方式创建：由用户或由程序来创建。Clouds 线程不像第 2 章所述的线程那样绑定于单地址空间。当一个对象的执行产生对另一个对象的调用时，线程暂时地离开该对象而进入新调用的对象。在调用的对象中执行完毕后，线程才返回到原来的对象中。对象调用可以是递归的或嵌套的，线程只有在它完全执行完整个操作时才终止，如图 6-2 所示。

线程允许在一个对象内并发执行，这样可以在一个对象中同时有多个线程。当多个线程存在于一个对象中时，它们共享对象的数据及对象的地址空间。Clouds 使用锁和信号量来支持对象内的并发控制。所有的对象都必须编写成支持并发执行的。当在执行过程中创建并发线程时就会指定并发的实际级别。

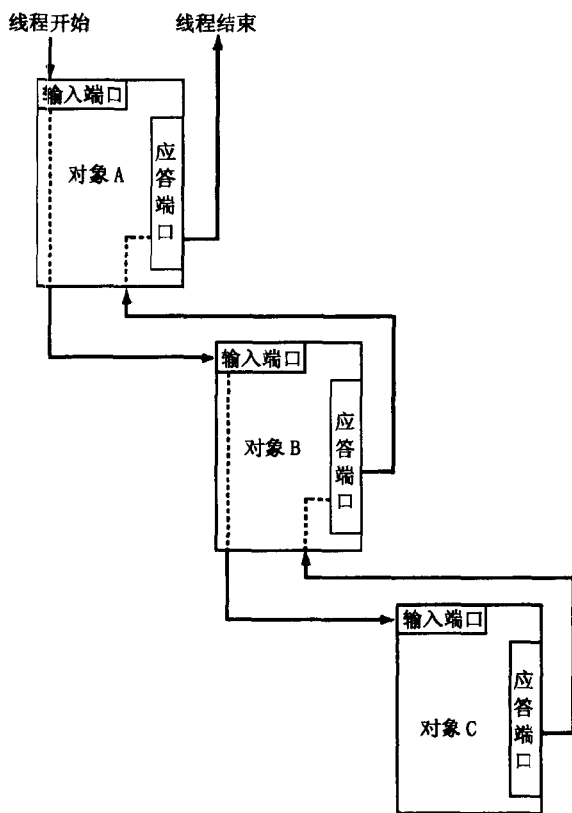


图 6-2 Clouds 的线程执行

6.2.3 Clouds 内存存储

在传统的系统中，持久存储用于存放文件，而与进程相关联的存储器中的信息是易失的。Clouds 的对象用来统一持久存储和内存，以创建一个完整的、持久的地址空间。该机制不支持消息和文件但可以模拟。对于共享数据的对象，则必须调用数据所在的对象。从内存管理器的角度来看，Clouds 的对象由映射到虚拟内存的多个段组成。所有的段都有若干物理

页大小，由称作分区的系统对象来维护这些段。分区负责段的创建、维护和存储。一个段属于创建它的分区。

6.3 Chorus V3 和 COOL V2

Chorus V3 (版本 3) 是一个分布式实时微内核操作系统，在 [LJP93] 中有该系统的相关讨论。它最初由法国 INRIA 研究所作为一个研究项目进行开发。从那时起，它慢慢地进化成一个遵循 CORBA 标准的商业化产品系列。具体地说，Chorus V3 包含一个面向对象子系统，这个子系统称为 Chorus 面向对象层 (Chorus Object - Oriented Layer, COOL) 它现在是遵循 CORBA 标准的分布式实时商业化产品 Chorus/COOL ORB 的一部分。在第 3 版作为商业化产品发布后，Chorus 于 1997 年下半年由 Sun 公司收购。从那时起 Chorus 就作为 Sun 公司 JavaOS 的基础，并由 Sun 公司单独冠以 ChorusOS 的名字在市场上发售。

COOL 子系统运行在 Chorus 微内核之上。Chorus V3 的一个重要目标是在进程控制和远程通信方面的系统响应，这对于实现分布式实时执行内核是非常关键的。

COOL 对象由两个段组成，分别是数据段和代码段。这个代码段作为方法的对象方法存放在代码段中，并由所有同一对象的实例共享。

在本节中，我们讨论 COOL V2 (版本 2)。COOL 体系结构由以下三部分组成。

1. 基层。
2. 通用运行时系统。
3. 语言运行时系统。

用户程序运行于语言运行时系统之上，我们将一一讨论这些层。

6.3.1 基层：COOL 内存管理

基层为用户进程提供一组服务，它跨越整个系统并延伸到 (基于进程的) 微内核。该层提供了 DSM 的面向对象的等价系统，并支持消息发送。Chorus DSM 在详细说明 4.4 中讨论。COOL 中的内存管理是基于集群的。集群是持久的，基层能定位非活动的集群。每个集群是一组由段组成的区域，集群的成员通常是属于同一个类的相关对象。该体系结构的上层软件确定实际的集群的分配，这些集群映射到任意的地址空间且在逻辑上是一组虚拟内存区域。分布式虚拟内存映射器管理虚拟内存和辅助存储器之间的关系。分布式虚拟地址空间称为上下文空间。上下文空间允许集群共享属于某特定上下文空间的执行线程。可以改变上下文空间的实际映射的绑定，这样在需要的时候能重新映射到另一组地址空间。这种方法允许一个集群同时映射到多个上下文空间，而无需所有的上下文空间都映射到物理地址中。这样当前未利用的上下文空间可以保持为未映射状态。

详细说明 6.2

一个 COOL 内存例子

在本例中我们使用如图 6-3 所示的两个节点：节点 A 和节点 B，如图 6-3 所示。有三个上下文空间：上下文空间 1 和 2 位于节点 A，上下文空间 3 位于节点 B。此外有三个集群，如前所述这些集群可以包含在多个上下文空间中，集群 1 映射到三个虚拟内存区域：

上下文空间 1, 2 和 3。集群 2 映射到上下文空间 1 和 3 这两个虚拟内存区域, 集群 3 也映射到两个虚拟内存区域: 上下文空间 2 和 3。

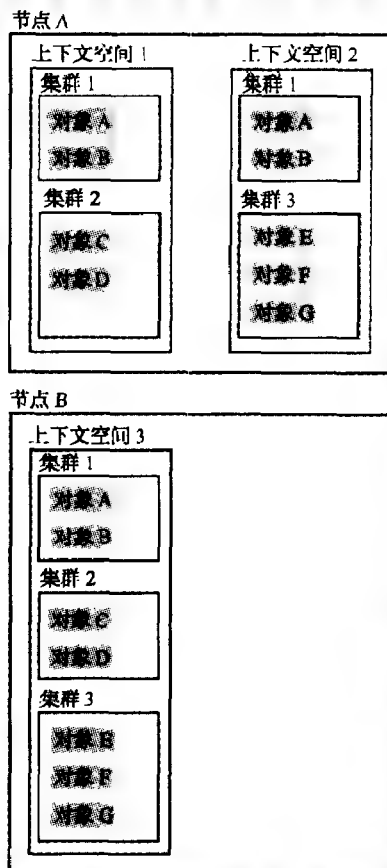


图 6-3 COOL 存储器

6.3.2 通用运行时系统层: COOL 对象

COOL 对象是在通用运行时系统层中实现的, 由状态和一组方法组成。这些对象为应用程序开发提供服务。有两种对象引用可用: 第一种是领域范围内持久的和全局惟一的引用; 第二种是在对象的当前上下文空间中有效的语言引用, 是对虚拟内存的引用。

通用运行时系统层的一个组件是对象管理组件, 对象管理包括以下操作:

- ◆ 对象创建。
- ◆ 动态链接。
- ◆ 动态加载。
- ◆ 透明调用 (映射到上下文空间和辅助存储节点)。

该层支持一个执行模型和一个语言级的模型, 执行模型将对象活动映射到 Chorus 的线程和作业中。当一个对象开始动作时线程开始执行, 线程由下层的微内核来支持, 作业用来对分布式执行进行建模。每个作业代表了单个的分布式应用, 具体地说, 作业组织一组上下

文空间，如详细说明 6.2 所述每个上下文空间能支持数个集群，而且每一个集群能包含多个对象。此外，集群能支持并发活动。并发在本地由同步原语支持，在非本地的并发由分布式令牌管理器支持。本地的同步原语包括信号量和多读者/单写者的锁，详见第 5 章。

语言层的模型设计成支持不同语言模型的多种语义，具体地说，部分实现了 ORB（对象请求代理）。这个模型支持持久性、调用及地址空间的重映射。一般来说，它允许该层在分布式环境中为多种语言以语言特有的方式运作。利用称为接口对象的代理服务可将本地的方法调用替代为远程调用。这种代理服务用打包和拆包参数的远程过程调用来代替语言通信原语，它允许并且在事实上实现了对对象的分布式执行。

6.3.3 特定语言运行时系统层

该层利用预处理器来负责将特定语言的对象映射到通用运行时模型，这样就实现了如 6.6.1 节所述的 ORB 的 IDL（接口定义语言）。程序员可以使用通用语言 COOL++（它是 C++ 的一个变种），或用标准 C++ 编程，并显式地利用通用运行时系统层的系统调用。使用 COOL++ 的主要优点是它遵循 CORBA（见 6.6 节）的规范，而成为一种透明的编程模型。

6.4 Amoeba

Amoeba 是基于处理器池的模型，允许处理器数目动态变化。Amoeba 的进程管理和存储管理已分别在第 2 章和第 4 章的详细说明中作了介绍，在本节中，我们集中讨论 Amoeba 的对象技术。Amoeba 的对象可以是客户机对象或服务器对象，对象允许多重继承。我们在 6.4.1 节中讨论 Amoeba 对象的标识和保护，在 6.4.2 讨论对象通信。要了解 Amoeba 系统的更多信息，请参阅 [MVTVV90, TaAn90, Tan95]。

6.4.1 Amoeba 对象的标识和保护

对象的标识是基于由以下四个基本字段组成的结构化的权能：

1. 服务器端口。
2. 对象编号。
3. 权限字段。
4. 校验字段。

服务器端口是对管理特定对象的服务器进程的编码表示。同 Clouds 的对象一样，Amoeba 的对象有一个随机选择的惟一名字。该名字以对象编号来表示，这个编号由目录服务映射为 ASCII 名字。可以对对象执行的操作（如读、写、执行）由权限字段的内容来规定。最后，权能也负责对象的保护，这由校验字段来实现。校验字段通过提供加密保护来预防他人猜测对象的权能，这种保护也允许权能管理以用户进程在内核外出现的情况。

6.4.2 Amoeba 的对象通信

Amoeba 的对象通信是通过远程过程（对象）调用来实现的。总的来说，有三种类型的远程过程调用原语：

1. do_operation,

2. `get_request`。
3. `send_reply`。

`do_operation` 原语由客户机线程发出, 用来从对象管理服务中发送请求, 要求对象执行操作。如果服务器愿意在某特定端口接受远程过程调用, 它就利用 `get_request` 原语来将此通告给系统。当服务器收到一个 `do_operation` 请求时, 服务器就执行该请求并通过 `send_reply` 将应答返回给客户机, 见图 6-4。同大多数操作系统一样, 这些基本的通信原语通过由其他的类继承使得在同一层中功能进一步增强。

消息在对象的端口被接收。端口是一个 48 位的数字, 只有服务及其客户机知道。公共服务(如文件系统)通告它们的端口, 而私有服务端口则保密, 因为知道了端口号就意味着允许使用服务。为了能够让所希望的请求使用服务, 除了端口号外客户方还必须拥有正确的权能。因此, 通信机制包括安全措施, 端口保护对服务器的访问, 以及权能保护对象。此外服务器能使用非对称加密(见第 11 章)来进一步增强安全性。

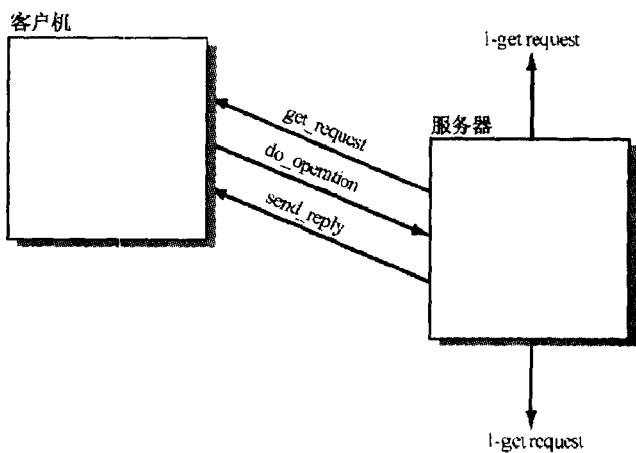


图 6-4 Amoeba 的对象通信

137

第 3 章曾讲过远程调用需要参数打包。在 Amoeba 中参数打包(和拆包)由接口语言编译器来完成。在打包完一个请求后, Amoeba 的传输机制就被调用来发送消息。消息由一个消息头和一个缓冲区组成, 缓冲区通常含有数据而消息头则有固定的格式。消息头中包括地址信息、操作码和其他依赖消息类型的信息。操作码用来选择服务特定消息的对象。

6.5 分布式组件对象模型

组件对象模型 (COM) 曾经是基本的 Microsoft 组件对象模型。用于分布式 COM 的分布式组件对象模型 (DCOM) 为分布式对象计算提供了一种编程模型、执行代码级的标准以及互操作标准。自从 Windows NT4.0 发布以来, DCOM 就可用, 它拥有用于 Windows 95 的版本并得到 Microsoft Internet Explorer 4.0 及 Windows 98 支持, 从一个名叫 Software AG 的公司可得到能应用于所有主要的 UNIX 平台的 DCOM 产品。虽然开始时由 Microsoft 发起, 但现在 COM 和 DCOM 已不再为 Microsoft 所专有, 而是由一个独立的 ActiveX 协会来负责对它们的管理。本节中的内容是从 Microsoft 的 NT 白皮书集 [Micr98] 中摘录下来的。

基本的 COM 模型拥有允许单独考虑逻辑单元的能力。它也允许灵活地执行代码组件, 可适应不同的配置和机器。COM 被广泛地看作是 ActiveX 中的核心技术, 任何支持 COM 组件的软件工具也自动地支持 COM 的分布式扩展——DCOM。有四种对 DCOM 特别有用的 ActiveX 服务器框架服务: 拥有回滚和恢复能力的事务; 可以在间歇性不可访问的网络上操作的、拥有可靠的存储-转发队列的排队系统; 易于与基于 HTML 的 Internet 应用集成的服务

138

器脚本；对旧产品系统的访问支持。DCOM 比 COM 更进一步，它使软件组件通过网络直接通信。DCOM 的一些突出的特点是：

- ◆ 传输中立：DCOM 能使组件间进行通信而无论它们是面向连接的或是无连接的，它支持 TCP/IP、UDP/IP、IPX/SPX、Apple Talk 和 HTTP。
- ◆ 开放式技术：DCOM 是一种开放式的技术，它可用于 UNIX、Apple、Windows 及某些遗留环境，但它还是最常用于基于 Windows 的环境。
- ◆ 通用 Web 浏览器和服务组件：自从 DCOM 包括 ActiveX 及 ActiveX 组件以来，它就能嵌入到基于浏览器的应用程序中，DCOM 使支持浏览器技术的分布式 Internet 应用成为可用的。
- ◆ 安全性：DCOM 能将基于认证的 Internet 安全机制（见第 11 章）集成到基于标准的应用中。
- ◆ 分布式计算环境（DCE）的 RPC 机制的扩展：DCOM 对 DCE 的 RPC 机制进行了扩展，扩展位于 DCE 的 RPC 的上层，它也是建议的 Internet 标准的一个主题。

基本的 DCOM 体系结构将应用设计和开发成能自动允许其将来进行分布和扩展的一种方式。一般来说，添加一个服务器到系统中比经常地升级系统开销更少和更加实用。这又与 DCOM 有什么关系呢？用 DCOM 设计的应用在部署时具有内在的灵活性，一个应用刚开始时可能作为集中式解决方案在单个服务器中部署，将来对应用的需求可能会超过服务器处理能力，通常就需要重写该应用程序或将服务器升级。对于 DCOM 的应用程序，其他的服务器可以加入到系统中并使不同的组件在新加入的服务器中运行。这样就极大地增强了应用程序的可扩展性，并极有可能减少工程维护费用，因为不需要修改应用程序。应用的实际可扩展程度依赖于应用程序的体系结构，当然如果应用程序当初未进行恰当设计，它可能不具有 DCOM 的这方面的优点。

为了与非本地的组件通信，DCOM 使用一种对应用程序完全透明的进程间通信机制。具体地说，DCOM 用一个网络协议来代替本地通信，因此，DCOM 为应用提供了位置的独立性和位置的透明性。

6.5.1 标记

DCOM 中的实例名字称为**标记**（moniker）。标记本身就是对象，它有着极大的灵活性。从数据库名字到服务器、统一资源定位符（URL）以及超文本标记语言（HTML）页面都是标记，它们可以用来标识对象的实例。标记含有必要的信息和逻辑以定位它所命名的对象当前正在运行的实例。一个命名的对象的标准接口称为 IMoniker，这个 IMoniker 接口可用来绑定到被命名的对象实例上。

一般来说，对象实例命名它自己。在命名过程中它们创建一个标记对象，以提供给有兴趣重连接到它们上的客户方，而且对象实例将它们的标记对象登记到 COM 库中。当前正在运行的标记都列入到运行对象表（ROT）中，标记使用 ROT 快速找到运行中的对象实例。当一个标记对象需要绑定到一个对象实例时，ROT 努力寻找一个匹配。当匹配找到时就返回指向存放在 ROT 中的相应活动对象的指针。如果没有运行实例或标记没有登记，标记就利用对象创建机制来创建一个未初始化的实例。在创建时标记用相应的机制存储该对象的状态。这个完整的过程对应用是透明的。

标记拥有对象持久状态的参考信息, 这种信息一般存放在一个文件或数据库中。对象可以向标记表明当存放这种状态信息时应激活什么。这样如果对象运行在存放该数据的同一个服务器上, 数据密集型操作就可以防止潜在的性能瓶颈。如果多个客户机需要访问某对象, 则该对象应当在一个可以被所有客户机访问的节点上执行。

标记也有覆盖它们的绑定操作的能力。具体地说, 可以通过编程来覆盖标记绑定操作中要用到的远程服务器的名字。这通过一个包含在标记内的可选的指向 COSERVERINFO 结构来实现。这个结构用于 CoCreateInstanceEx 结构以创建对象实例。

文件标记是 COM 的一部分, 其中封装了用于对象实例化的文件协议。它可以用于将 COM 文件标记映射到 URL 文件协议, 访问在本地文件系统中的持久数据, 将文件协议初始化为访问已知的文件及作为确认存储的文件类型的检查机制。URL 标记封装了 IP 协议, 例如 http、ftp 和 gopher。

6.5.2 远程方法调用

当客户方需要调用位于另一个地址空间中的对象时, DCOM 执行远程方法调用。如前所述, DCOM 的接口描述语言 (IDL) 是构建于 DCE RPC 标准 IDL 之上。在 RPC 中通常将所有参数信息打包到一个标志存储缓冲区中 (见第 3 章), 接收方通过参数拆包过程重新创建堆栈内容来重构参数信息。当远程方法调用返回时, 任何返回值和输出参数需要再次从本地堆栈中打包然后发回给客户方, 客户方将参数拆包就完成了—次完整的通信过程。客户方执行远程方法调用的代码称为代理, 而服务器方的代码称为桩程序。

140

另外一个未包括到 DCE RPC 中的数据类型是接口指针。这种类型的指针可以作为 CoCreateInstance 的返回值或方法调用的参数出现。为了处理这种新的数据类型, 创建了能在接口的所有方法中都能处理该数据类型的代理/桩程序对。传统上在 COM 和 DCOM 的大多数领域中, 对接口指针的打包过程可以扩展或通过使用定制的代理/桩程序对来覆盖。

6.5.3 资源回收

用来控制已知对象的生命周期的关键机制是引用计数。一个对象的引用计数通过使用 AddRef 来增加, 而由方法 Release 来减少。希望所有远程客户程序正常终止并执行一个 Release 操作是不现实的, 因此使用 Ping 机制。Ping 是用来检测客户程序非正常终止的常用机制。每个引出对象都有一个 pingPeriod 时间值和 numPingsToTimeOut 计数, 这些值组合起来确定了称为“ping 周期”的总的时间。如果已经过了“ping 周期”时间而没有为给定的对象 ID 一次“ping”操作, 则认为远程引用过期了, 引用计数像进程正常地终止一样被减少。当一个对象没有进行引用的话就会回收该对象所用的资源。

6.5.4 DCOM 中支持的线程模型

COM 和 DCOM 以在本地操作系统中多线程的方式 (见第 2 章) 运行。由于 DCOM 的进程间通信需要 DCE RPC, 而 DCE RPC 要求能在任意线程中处理 RPC 消息, 因此对 DCOM 也有同样的要求。COM 和 DCOM 的编程模型随当前应用的需要扩展, 这样应用程序开发人员需要提供完全的自由线程调用, 此外 DCOM 为单线程提供了方法调用的自动同步。

关于线程, DCOM 使用了套间模型。从参数打包方面来看, 可以将每个线程看作一个独

立进程。对本线程的访问是直接访问而对其他线程的访问是间接访问，间接访问可能是通过代理/桩程序对。对象可以支持以下三种线程模型：

1. 单线程套间，只有主线程：在这种模型中，所有对象的实例由与该对象相关的同一个单线程创建。

2. 多线程套间：每个实例被束缚于一个单线程，不同的实例可以由不同的线程来创建。

3. 多线程套间：实例可以在多个线程中创建且可以由任意线程调用。

任何作为本地服务器来运行的对象控制着它的对象的套间类型。本地服务器通过一个或多个线程上的 CoInitializeEx 来登记该类或多个类工厂，从相应线程到相应线程套间模型。当创建一个对象时，COM 需要知道该对象是否与其父对象的线程模型兼容。如果不兼容，那么 COM 就尽力将该对象加载到与之兼容的另一个套间中。若兼容，则相应的代理/桩程序就会分别放置于客户方和该对象间以使它们之间能够通信。由于 DCOM 的目标是让应用程序开发人员具有全面的灵活性，因此它能够定制线程模型。

6.5.5 DCOM 的安全策略

与所有支持分布式计算的系统一样，DCOM 也必须注重其内部的安全性问题。基于这点考虑，设计 DCOM 来处理以下四方面的安全性：

1. 访问安全性：保证一个对象只允许由具有相应权限的对象成功调用。

2. 激发安全性：保证只允许合适的对象在新进程中创建新对象。

3. 身份：对象依靠什么原则来鉴别它自己。

4. 连接策略：包括消息何时可以被修改，以及消息能否被另一个对象截取，此外对象身份的鉴别也是属于该类型。

访问安全性可以在对象级、方法级、参数级或在更高的进程级上实现。访问安全性甚至可以被设置来向不同的调用者提供不同的安全检查。DCOM 的对象能够设置它们的安全策略，并允许创建新对象而无需显式的安全信息。

激发安全性是在 COM 库中实现的。这些库检查调用者所请求操作的权限，这些权限信息设置在对象外面的注册表中。

就提供安全身份而言，DCOM 支持多种安全措施，包括从对于大量用户应用来说很繁重的对象级安全措施到虽然方便但可配置性稍差的群组级安全措施。安全身份可能会由于每个账号不只限于给单个用户而进一步复杂化，这样使基于用户的安全访问变得更加困难。每种方法都有优点和缺点，DCOM 包含多种方法以便开发人员灵活使用。

对于连接安全性，DCOM 又一次为应用程序提供多种选择，这样开发人员可以对应用选择最合适的安全级别。源地和目的地的网络连接数越多，通信被截取的可能性就越大。安全等级越高，就需要越多的开销来执行所要求的通信。实际安全等级可以动态地选择。一个最好的预防措施是利用 DCOM 支持加密数据传输的能力。用这种方法就可以确保内容的保密性和完整性。

当使用访问检查、激发权限检查和数据保护机制时，需要某种方法来确定客户方的身份。有几种安全机制可用，这些安全机制返回惟一的会话权标作为整个连接期间的鉴定。

由于常常要求对象代表它们的父对象执行操作，安全措施也需要结合到四种不同的模拟级别中来。第一等级是匿名 (SecurityAnonymous)，它防止得到调用者的身份。第二等级

是身份鉴别 (SecurityIdentification), 它允许对象检测调用者的安全身份, 但对象不能模仿调用者。第三等级是模仿 (SecurityImpersonation), 它允许对象模仿并执行本地操作, 但对象不能代表调用者调用其他对象。第四等级是委托 (SecurityDelegation): 对象可以模仿调用者, 而且对象可以使用调用者的安全身份执行方法调用。在 Windows 2000 操作系统中, 允许 DCOM 能进行全面的系统访问并可以利用基本的 Windows 2000 安全框架 (在第 12 章讨论)。应当注意到全面的系统访问可能会使一个参与系统的可信度和安全性受到破坏。

6.6 CORBA 概述

公共对象请求代理程序体系结构 CORBA 由对象管理组 (Object Management Group, OMG) 所制定。CORBA 目的是为在异构分布式环境中的应用提供互操作性。它有时也称为中间件, 因为它不执行对于操作系统所要求的最低层的功能。虽然 CORBA 必须在传统操作系统之上, 但是它能执行传统操作系统领域为特定的分布式环境设计的许多操作。CORBA 最初由数字设备公司、HP 公司、HyperDesk 公司、NCR 公司、对象设计公司 & SunSoft 公司的代表组成的委员会所定义。参与制定 CORBA 2.1 规范的公司和厂商超过 750 家, 该版本的规范在 1997 年八月被采纳。它在 [OMG97] 中定义。在本节中, 我们只对一些关键的概念作简短的概述, 因为完整的正式规范将近本书字数的两倍。最后, 需要指出的是本节中的信息和图解是基于 CORBA 规范 2.1 [OMG97] 和规范 2.2 [OMG98a], 这些使用得到了 OMG™ 的许可。

143

如前所述, CORBA 的设计是全面的但又不失灵活性。CORBA 的贡献和目标是简化并包含有异构系统或语言的异构环境下的计算成为可能。CORBA 的核心是对象请求代理程序 (ORB), 这将在 6.6.1 节中讨论。CORBA 其他的主要特性是对象适配器的定义, 见 6.6.2 节。CORBA 的消息模型将在 6.6.3 节中讨论。在 6.6.4 节讨论 CORBA 标准的遵从。最后 CORBA 对 COM 的映射将在 6.6.5 节介绍。

6.6.1 CORBA 的 ORB

如图 6-5 所示 ORB 提供了透明的接口, 特别是接口独立于对象的位置及实现语言。这样 ORB 允许 Smalltalk 的对象与 C++ 的对象通信并一起工作, 即使它们在物理上位于地球两端。ORB 负责处理其中所有的转换, 通过与对象适配器相接的通用接口定义语言 (IDL) 来实现。接口由具体的操作及参数组成, 有三种类型的接口:

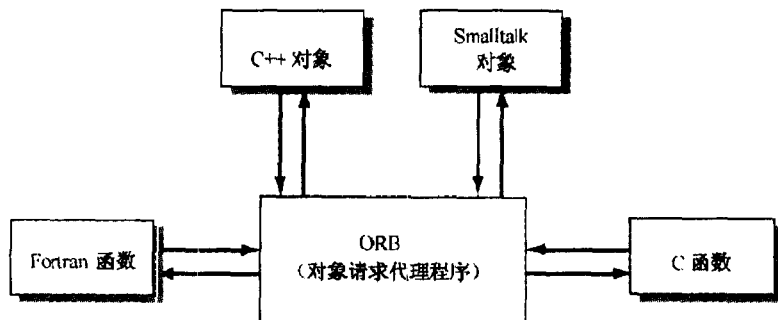


图 6-5 ORB 的功能

1. 通用型：通用型接口用于对所有 ORB 实现都相同的操作。
2. 对象类型：这种接口只应用于特定类型对象所特有的操作。
3. 对象样式：该接口只用于对特定式样的对象实现所特有的操作。

6.6.2 CORBA 的对象适配器

CORBA 2.1 [OMG97] 规定了对象适配器的以下职责：

- ◆ 对象引用生成。
- ◆ 对象引用解释。
- ◆ 方法调用。
- ◆ 安全交互。
- ◆ 对象及实现的激活和撤消。
- ◆ 对象引用到对象实现的映射。
- ◆ 所有对象实现的注册。

对象适配器能将职责委托给 ORB 内核。它们负责维护自己的状态。当处理调用时，它们事实上是在跟使用 IDL 的、称为框架的组件进行交互。一个对象实现收到请求，以及框架如何安排进 ORB 体系结构的例子如图 6-6 所示。IDL 是一种与 C++ 非常类似的高层对象语言，对象适配器的每个实例都必须在所有的 ORB 实现中有同样的接口。利用对象适配器，ORB 为面向对象和非面向对象语言提供了编程语言映射。这样 C 和 Fortran 程序可以利用 CORBA ORB，跟 C++ 和 Smalltalk 程序协同工作。

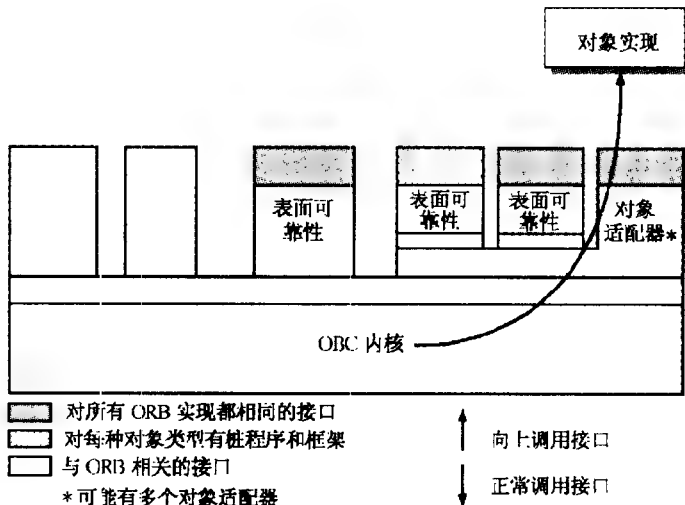


图 6-6 对象实现收到请求

虽然基本对象适配器（BOA）满足所有这些职责，但并非所有的对象适配器执行每一个职责。例如设计的库对象适配器是用来支持带有库实现的对象。由于假定这种对象是客户机程序，库实现不支持对象激活和身份检查。又如数据库对象适配器不需要提供状态信息，因为这些信息会在面向对象数据库中永久存储。

在 CORBA 2.2 之前, BOA 是惟一定义的标准对象适配器。结果, 每个厂商都实现各自的 BOA, 在这个过程中显示出了规范的含糊性及缺点。可以想象这导致了自定义的专有 BOA 实现。将损害 ORB 之间的互操作。CORBA 2.2 可通过定义可移植对象适配器 (POA) 来缓解这个问题。

服务器对象在实现一个使用 POA 的对象之前必须先获得一个 POA 对象。根 POA 就是由 ORB 管理的著名 POA 对象。这个根 POA 提供给使用 ORB 初始化接口的应用。只要默认的策略合适, 应用程序开发人员就可以使用根 POA 来创建对象。总共有七种与根 POA 有关的策略:

145

1. 线程策略: 该策略支持包括多线程 (见详细说明 6.3) 在内的线程选择。

2. 生命期策略: 除了以前支持的持久对象以外, 又有支持短暂对象的能力。短暂对象的生命周期受其父 POA 生命期的限制, 而持久对象的生命期在多个服务器中间持续。这种新的可选方案对于清除孤儿对象, 以及那些父对象已经终止从而不需继续执行的子对象特别有用。

3. 对象 ID 惟一策略: 该策略对于惟一地标识每个对象很必要。

4. ID 赋值策略: 对象的 ID 赋值可以从应用程序或 POA 中获得, 这将产生一个 `system_id` 值。

5. 服务工具保留策略: 如果 POA 有保留策略, 它就从请求中得到对象 ID 值, 调用相关联的对象。如果 POA 有非保留策略, 则 POA 将使用默认的服务工具或服务工具管理器来定位一个活动的服务工具。服务工具只在单个请求期间内才是活动的, 没有与服务工具-对象关联的记录保留下来。

6. 请求处理策略: 对于给定的对象 ID, 若没有服务工具与它关联或定位, 则 POA 就使用 `USE_DEFAULT_SERVANT`、`USE_SERVANT_MANAGER` 或 `USE_OBJECT_MAP_ONLY` 策略来请求处理。如果以上策略都不存在, 则将引发相应的异常。这就提供了将对象请求调度给多个对象实例的能力, 必然会增强系统均衡分布负荷的能力。

146

7. 隐式激活策略: 它允许 POA 用指明对象可由隐式激活的策略来创建。一个隐式策略需要有一个保留策略和一个 `system_id`。POA 提供了支持隐式激活的操作。POA 除了有执行隐式对象注册的能力外, 还能执行显式注册。这允许在系统内部发出请求之前, 对象就创建可以通过 POA 来接收请求的实例。

详细说明 6.3

CORBA POA 线程模型

当应用程序在 CORBA 框架内创建一个 POA 时, 它可以在两种线程模型中选择其中的一种用 ORB 实现。如果选择了单线程模型, 则单个线程处理从那个 POA 发送来的每一个对象请求, 这种选择的结果之一是产生顺序处理。同样地, POA 也可以选择由 ORB 控制的模型, 在这种情况下由 ORB 选择线程模型, 因此若低层的运行环境支持多线程则可以实现多线程的模型。由于有多线程的可能性, 所有以这种方式实现的 POA 都应当能够处理包括并发、重入及可能出现的多个对象实例在内的多线程。

6.6.3 CORBA 的消息模型

传统的 CORBA 消息提供了三种用于消息调用的模型：第一种模型是同步模型，客户方调用操作后就阻塞起来直到操作执行完毕；第二种模型是阻塞模型，客户方调用操作后继续执行下去，之后通过轮询来获取应答或阻塞起来等待应答，它只用于动态调用；第三种消息模型是非确认通信，该模型保证尽力完成通信，但发送方不等待或检查或期待一个应答或确认，设计这种模型是为了使 CORBA 支持 UDP。

6.6.4 遵从 CORBA 标准

147 对于一个遵从 CORBA 标准的系统，它必须符合 CORBA 内核的规定，以及至少提供一种映射。在标准中指定的语言是 C, C++, Smalltalk, COBOL, Ada 及 Java。对于每种语言在特定位置的实现，它们必须按照 CORBA 规范映射到该语言，称为遵循 CORBA。换言之，一个系统要么完全遵循 CORBA，要么不遵循 CORBA。如果一个厂家支持 C++，它的 ORB 就必须遵守标准中规定的 OMG IDL 到 C++ 的绑定。

6.6.5 CORBA 到 COM 的映射

CORBA 2.2 规范在第 16 节给出了数据类型从 CORBA 到 COM 映射的信息。标准间的互操作能力对于在异构环境中成功构建和部署分布式系统是很重要的，因为有可能需要将两种标准合并。这不是一件容易的工作，例如 CORBA 和 COM 都利用接口标识符来唯一地标识对象接口。通过标识符，客户方代码能够获取包括对象的其他接口信息在内的对象信息。若 CORBA 和 COM 都给接口指定一个文本的名字，虽然便于记忆但都不能保证它的惟一性。这样 CORBA 使用能够惟一地标识接口的 RepositoryId 来标识接口，而 COM 使用被称为 IID 的结构来标识接口。CORBA 和 COM 处理异常的方式各不相同。CORBA 利用异常来报告出错，并且允许复杂的结构通过通信交换异常信息。而 COM 仅使用操作的返回结果 HRESULT 的类型来报告错误，该结果随后转换成 16 位的代码。不幸的是，不可能将 CORBA 的异常完全地映射到这些代码。虽然 CORBA 对于单向操作指定了求最多一次语义（见第 3 章），但是 COM 却不保证最多一次语义。这些只是这两种标准之间的不同之处，以及互操作时遇到的困难的几个例子。尽管存在这些困难，互操作仍然是可能的，并且随着时间的迁移，它将会变为一种简单的任务。

6.7 小结

在本章中，我们讨论了设计操作系统的对象方法。可以看到在那些选择基于对象的操作系统中如何处理通信、线程、同步和存储管理等这样一些概念。

并非所有的对象方法都允许继承和多重继承。当允许对象继承时必须要小心。尽管对象方法存在缺点，但该方法还是成功的，并且关于这个主题目前仍在进一步研究。DCOM 和 CORBA 也在快速发展，因为它们定义了用于分布式计算的有效模型。除了基本 CORBA，也有用于特定领域的 CORBA 体系结构，例如用于医疗领域的 CORBAMed 和用于电信业的 CORBAtelecoms [OMG98b]。市场上有基于 CORBA 的产品，比如实现了 ORB 思想的 Chorus COOL。DCOM 一旦从 Windows 平台诞生后，就广泛地应用于 Windows 和 UNIX 平台，它也是分布式领

域的主导对象技术。

虽然我们已经讨论了三种操作系统和两种主要的协议，参考文献部分还有其他基于对象的操作系统和相关工作的信息。Chorus COOL 和 Amoeba 都支持多处理器（并行）的分布式系统。Chorus 微内核也支持实时应用，虽然这种支持与它的基于对象设计无关而是在它的调度算法中使用了优先级。随着这些系统的成功和面向对象的发展趋势，将会有越来越多的面向对象操作系统不仅是作为研究项目而且也会作为商业化的产品被开发出来。

6.8 参考文献

关于基于对象的实时操作系统的概述见 [AKZ96]。某些与本章内容有关的研究论文包括 [ARJ97, BSCEFHL93, CIRM93, DLAR91, GhSc93, HaSe98, Hen98, Her94, Kim97, Lin95, LJP93, LoKi97, Micr98, MVTVV90, OMC97, OMC98a, OMC98b, OSH95, Sch98, Sie98 以及 Vin98]。

以下提供了在 Internet 上的一些关于基于对象操作系统的链接。CORBA 的主页在 <http://www.omg.org/omg00/wicorba.htm>。关于 CORBA 的最新的规范、白皮书和参考书籍的列表见 <http://www.omg.org/store/pub.htm>。Amoeba 的主页在 <http://www.am.cs.vu.nl>。Chorus 的主页在 <http://www.Chorus.com/>。Choices 是另一种面向对象的分布式操作系统，带有出版物的主页在 <http://choices.cs.uiuc.edu/choices/index.html>。关于 SUN 公司的面向对象操作系统 Spring 的文章见 Spring 的论文主页在 <http://www.sun.com/tech/projects/spring/papers.html>。对 DCOM 在所有主要的 UNIX 平台有商业化的实现 Software AG 公司的主页在 <http://www.sagus.com>。

习题

- 6.1 允许多重方法的构造器的优点是什么？它会引起哪些困难？
- 6.2 与简单继承相比，多重继承的优缺点各是什么？
- 6.3 试阐述在分布式环境中使用基于对象的模型的优缺点。
- 6.4 在第2章中，我们讨论了各种与迁移有关的问题。请问使用对象作为迁移单位的优缺点各是什么？
- 6.5 CORBA 是有助于迁移还是妨碍迁移？请说明原因。
- 6.6 请阐述在分布式环境中使用 CORBA 的优点。
- 6.7 为什么 Clouds 只允许通过对象的入口点来访问对象？
- 6.8 为什么 Clouds 中的数据共享只能通过对对象的调用来实现？
- 6.9 Clouds 的段可有多多个页大小的优点是什么？
- 6.10 为什么对于 COOL 来说允许上下文空间重新映射很有必要？
- 6.11 本章讨论的所有操作系统都支持持久性。为什么在分布式系统中持久性是一个非常重要的特性？
- 6.12 请阐明由 COOL 的通用运行时系统层支持的代理服务的优点。
- 6.13 在 Amoeba 中，为什么端口的知识不能够充分保护 Amoeba 服务？试说明一种可以不需要客户方的权能的情形。
- 6.14 Clouds 和 Amoeba 都提供全局惟一名字。全局惟一名字的优点是什么？如果不使用全局惟一名字则可能会出现什么问题？

第 7 章 分布式进程管理

在本章中，我们将讨论分布式环境中有关进程管理的各种问题，包括对第 2 章中介绍过的一般进程管理的扩展。具体地说，在 7.1 节中介绍分布式调度算法的选择。7.2 节讨论各种调度算法，对每种算法都举出了一个例子。在 7.3 节讨论与协调者选举有关的问题，协调者是一个在使用集中式服务器的分布式系统中的进程，对各种问题的所有解决方案来说都是必要的。7.4 节介绍与管理进程有关的问题，重点讨论分布式系统中父进程由于系统崩溃而终止时子进程的有关问题。

151

7.1 分布式调度算法选择

Casavant 和 Kuhl [CaKu88] 为各种类型的调度算法的分类提出了一种分类法。我们将讨论 7.1.1 至 7.1.4 节中各种调度算法的选择。每种选择都能用来对调度算法分类。图 7-1 中介绍的调度决策图反映了调度算法的各种选择，详细说明 7.2 将概述全局调度器的算法选择。

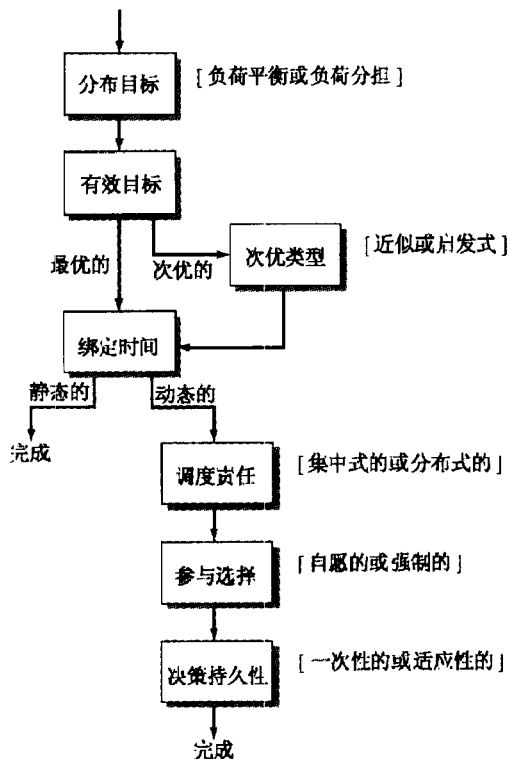


图 7-1 调度决策图

7.1.1 调度层次

分布式系统必须对调度的层次作出选择，这种选择在集中式系统中无效。在分布式系统

中,对进程的调度有两级层次:本地调度和全局调度。本地调度涉及到将一个进程分配到一个具体的处理器上,而全局调度则处理选择在什么地点上的哪个处理器来执行已知的进程。全局调度在有的文献中也称为处理器分配,全局决策必须在本地决策之前作出。在分布式环境中,两种选择都必须在某一时间内作出,但没有必要在同一时间内作出。在本章中,我们只讨论全局调度,与本地调度有关的问题请参阅第2章。

152

7.1.2 负荷分布目标

如第2章所述,有两种负荷分配目标。一个目标是负荷平衡,它是在整个分布式系统中各处理器间努力维持相等的负荷。负荷平衡需要相当大的开销和不断的系统性能评估。第二个目标更常见,是简单的负荷分担。负荷分担努力防止任何节点过载。与负荷平衡相比负荷分担是一个更容易达到的目标。图7-2描述了如第2章所述的负荷分配的三种状态:

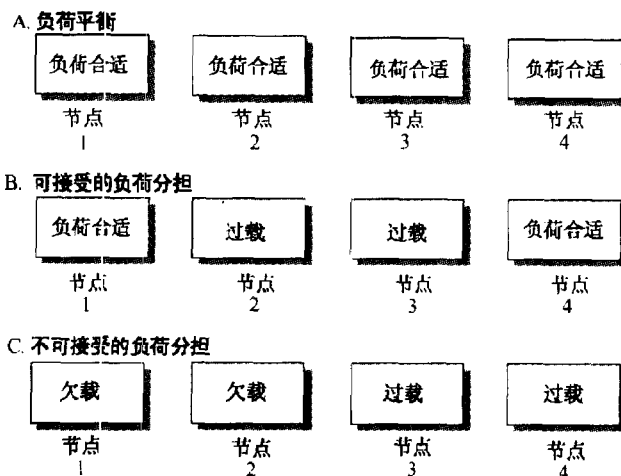


图 7-2 负荷分布目标

7.1.3 调度的有效目标

调度算法可以提供两级调度目标:最优调度方案和次优调度方案。对于最优调度算法,调度者必须掌握所有竞争进程的状态及相关的信息。最优化可以用完成时间、资源利用率、系统吞吐量或它们之间的任何组合来衡量。最优方案计算在超过两个处理器的情况下由于NP困难^①的问题而不可行,所以常常使用次优方案。次优方案分为两类:近似的方案和使用启发式方法的方案。详细说明7.1介绍了用来确定最优调度算法的基准:

153

详细说明 7.1

调度的有效基准

有许多基准用来确定和衡量调度的有效性,调度算法可以使用以下一个或多个基准。

① NP 困难问题是不可计算的,它没有已知的多项式时间算法存在。

一个最常见的基准是通信开销。使用这种基准的调度算法考虑用于向某给定节点迁移进程的时间开销。更重要的是它也必须考虑分配到特定进程位置的进程间的通信开销。当进程不需要同其他进程进行通信时，这种开销很小。若执行计算的过程中该进程与需要通信的其他进程位于同一个节点的话，开销也会很小。

执行开销是另一种流行的基准，该基准反映了将进程指派到给定节点的基于环境的开销。如果该环境与进程原来的环境一样，则执行开销为零。但如果给定节点运行不同的操作系统或使用不同字符串编码模式、不同的0表示方法和尾数表示方法等不同体系，则由于需要的转换量很大使得执行开销将会非常之大。

资源利用基准是根据分布式系统中各节点当前的负荷来确定将进程调度到一个恰当的位置。资源利用基准可以是基于节点的负荷状态，一个更加精确的基准可以包括负荷索引以提供比两级或三级状态更广的范围。资源利用基准也可以资源队列长度、内存使用情况或任何反映给定节点的资源状况为基准。

次优的近似方案

次优的近似方案常常利用同最优方案一样的算法，但不进行求出最优调度方案所需的所有步骤，而是通过限制搜索空间来尽可能快地找到一个非常好的调度方案。为了确定一个可接受的次优调度方案，必须对依据某种基准次优方案将产生怎样的结果有一些概念了解。此外次优方案必须适合于智能的快捷方法，这些快捷方法可能会使用启发式方法或先模仿最优方案而后使用启发式方法的近似方案。

次优的启发式方案

次优的启发式算法使用经验法则或直觉来进行调度。这些启发式方法也许不能被证明，并可能在某种环境下会偏离其出发点或完全错误，但一般认为它们能以可接受的方式发挥作用。以下是一些可能的启发式调度方法：

1. 需要大量进程间通信的相关进程应当位于邻近的位置，一般是位于同一节点。
2. 更改共享文件的多个独立进程应当位于邻近的位置，一般是位于同一节点。
3. 带有少量优先关系或无优先关系的可分割进程很容易分布，就将它们进行分布。
4. 如果某个系统的负荷已经很重，不要再调度其他进程到该节点。

以上规则如同所有的启发式方法一样，可能不会直接地或定量地影响系统性能，但这些基于直觉的规则能够帮助提高系统的性能。近似算法可以利用启发式方法来得到快捷方法，而启发式次优调度者在整个的调度算法中使用启发式方法。

7.1.4 处理器绑定时间

该项确定什么时候调度算法，以及决定进程将在何时何地执行，这种决定涉及到将进程绑定到某个特定的处理器组，即带有一个或多个紧耦合处理器的内存单元。调度算法可以提供**静态绑定**或**动态绑定**。在静态调度算法中，每个进程的可执行映像链接时已静态地指派给特定的处理器组。与静态绑定算法相反，当可执行映像创建时动态调度算法不决定进程在何时/何地运行，而是可执行映像以可以重定位的方式来创建，这样允许在以后某个合适时间里绑定到特定处理器（和位置）上。动态调度算法也必须确定调度者的责任和决定的永久性。

调度者责任

调度者责任可以由集中式或分布式的方式决定。该调度者决策可以应用到分布式系统和并行系统中，在分布式系统中更为常见。此外，调度者责任只应用于使用动态绑定的调度者中，它确定调度责任是否由在单处理器上的集中式服务器来履行，或者调度算法是否物理分布在多个处理器（可能在不同位置上）中。如果调度算法是集中式的，则作出所有调度决定的处理器成为系统中的关键部件。若调度处理器失效时没有备用处理器来接管它的工作，则当该集中式服务器崩溃或不可用时整个系统就会停止。

分布式的、动态的全局调度算法也必须对参与选择者作出决策。参与选择者包括自愿或强制选择参与者。在**自愿调度算法**中，每个本地的调度者对它的资源及它们的利用有更大程度的自治性。处理器不会也不能被迫参与或协作。相反，**强制调度算法**强迫每个处理器有它的分量，每个处理器被迫执行它的调度任务以满足系统范围的性能目标。不管调度者责任和参与状况如何，调度算法可以选择使用最优的、次优启发式的或次优近似的方案。

调度者决策的永久性

动态调度算法可以使用永久性分配（即所有的分配都是一次性的分配）或允许动态的重新分配。一旦调度决定作出后一次性分配是不可改变的，而动态重新分配允许在进程开始执行后迁移进程和重新分配。**一次性分配调度者**在进行分配时经常依赖用户信息，而使用动态重新分配的**适应性调度者**利用进程在执行时创建的信息来改变原先的调度决策和迁移进程。这样动态重新分配允许调度者不断地适应。当然，经常的改变由于迁移开销（见第2和第4章）会降低系统性能。虽然某些系统只使用其中一种方法或其他方法，但是大多数调度算法不是固定为一次性或适应性，而是可以使用任何一种方法。

156

详细说明 7.2

全局调度算法

以下是各种类型的全局调度者，是前面介绍的各种调度算法设计而产生的。这些调度者每个都可以用于负荷平衡或负荷分担，这样总共有42种可能的类型：

- ◆ 最优静态调度者。
- ◆ 最优一次性集中式动态调度者。
- ◆ 最优适应性集中式动态调度者。
- ◆ 自愿最优一次性分布式动态调度者。
- ◆ 强制最优一次性分布式动态调度者。
- ◆ 自愿最优适应性分布式动态调度者。
- ◆ 强制最优适应性分布式动态调度者。
- ◆ 次优近似静态调度者。
- ◆ 次优启发式静态调度者。
- ◆ 次优近似一次性集中式动态调度者。
- ◆ 次优启发式一次性集中式动态调度者。
- ◆ 次优近似适应性集中式动态调度者。

- ◆ 次优启发式一次性集中式动态调度者。
- ◆ 自愿次优近似一次性分布式动态调度者。
- ◆ 强制次优近似一次性分布式动态调度者。
- ◆ 自愿次优启发式一次性分布式动态调度者。
- ◆ 强制次优启发式一次性分布式动态调度者。
- ◆ 自愿次优近似适应性分布式动态调度者。
- ◆ 强制次优近似适应性分布式动态调度者。
- ◆ 自愿次优启发式适应性分布式动态调度者。
- ◆ 强制次优启发式适应性分布式动态调度者。

7.2 调度算法的方法

在上一节中，我们讨论了关于调度算法的各种选择，并枚举了基于这些选择的各种调度算法类型。本节我们将介绍调度算法的各种方法，每种方法都是先进行介绍，随后再详细地讨论使用该方法的一个调度者例子。以下是一些调度者类型的示例和它们所对应的章节：

- ◆ 7.2.1 节：负荷分担、次优、启发式、一次性的、集中式的动态调度算法。
- ◆ 7.2.2 节：负荷平衡、次优、近似、集中式的动态调度算法。
- ◆ 7.2.3 节：负荷分担、全局、次优、启发式的静态调度算法。
- ◆ 7.2.4 节：负荷分担、强制、次优、启发式、适应性、分布式的动态调度算法。

7.2.1 使用点数方法

每个系统的使用点数的方法可以维护和利用起来以保证对分布式系统的公平使用。这种方法一般涉及到一个集中式服务器，最好是用在小型的分布式系统中。该集中式服务器维护着一个使用表，该表对每台加入到系统中的计算机含有一个入口。如果某台计算机请求并利用不在本地的资源，例如利用一个远程处理器来调度一个进程，则它就支取使用点。如果一个参与者允许另一个参与者使用本地资源，那么就会减少该节点的使用点，如图 7-3 所示。一个慷慨的节点与大量地使用外部资源的节点相比，

使用点的点数要少一些。给定节点的总的使用点数用来确定调度，使用点数较低的节点总是在使用点数较高的节点之前被调度来提供非本地服务。

使用点比较简单，因为无论是使用外部资源支付一个使用点，还是允许外部进程在本地执行时收取一个使用点，每个动作都认为是一个使用点。此外还可以实现一种更加复杂的受

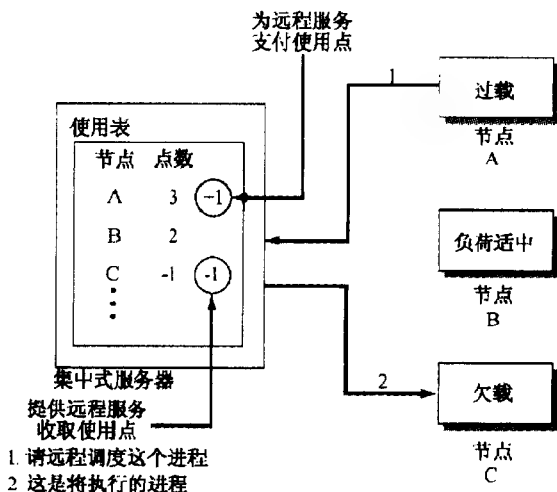


图 7-3 使用点数方法

经济学影响的模式,这种方法也称为**经济使用点**。在这个使用点方法的变种中,利用外地资源的支付是基于该系统的负荷。

在负荷很重的系统中利用外地资源比在负荷较轻的系统中利用外地资源支付得更多。当本地系统负荷很重时允许一个外部进程在本地执行会得到更大的点数。在任何一种情况下,向一节点收取的点数等于被使用的系统得到的点数。每个节点维护着一个如图7-4所示的当前报价表。建立基于经济学的使用点可以严格地确定被利用资源的负荷索引,或者可以基于投标模式。当将投标模式结合到该算法中时,支付的点数(仍然等于收取的点数)是可

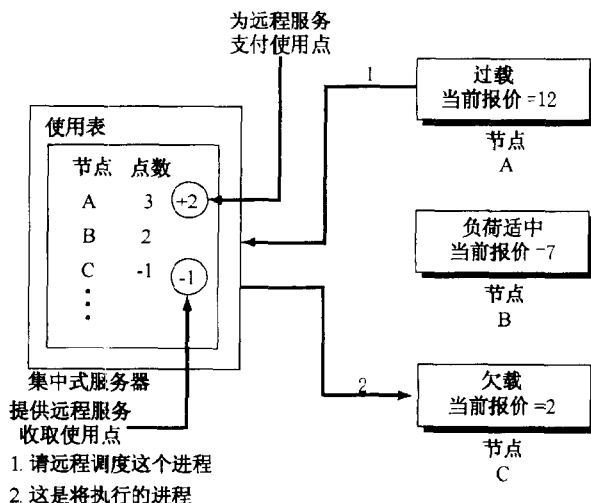


图 7-4 经济使用点

协商的,这样系统可以到处寻找最佳的交易。投标模式的思想最初是由 Ferguson、Yemini 和 Nikolaou 引入的 [FYN88]。

一个基于简单使用点的调度算法的例子如论文 [MuLi87] 所述,他们的算法提出了一个次优、启发式、一次性、集中式的动态调度算法,称为上-下算法。除了允许其他参与者在本地资源时减少使用点外,参与者也会因为未满足的资源使用要求而临时地减少点数。如果当收到请求时集中式调度服务器任何参与者没有拥有可用的资源,进程调度请求不能满足。如果系统没有未解决的请求且资源是可用的,则所有节点使用点也会减少。这种减少可以是每次出现时的一次性事件,或存在这种状态时允许按预定时间帧的多次调整。例如,所有系统在每次存在这种状态时减少一点或处在这种状态的每分钟就减少一点。上-下算法允许使用点数小于零,这样就可允许当一个慷慨节点需要大量帮助来执行进程时,积累信用以用于将来某个时间。

159

7.2.2 图论方法

这种方法的核心在于图论,具体地说在于获取图中顶点的最小割集或计算图的最大流量。我们首先介绍称为**最小分割 (min-cut)**的顶点的最小割集问题,这种方法利用了一个**指派图 (assignment graph)**。在指派图中进程和它们的可能指派表示在一个无向图中。该图的顶点如果记为 p 则代表一个进程,若记为 l 则代表一个节点,该图中没有标示优先关系。如果两个 p 顶点之间有一条边,则说明被表示的两进程互相通信,应当被指派到同一节点。如果一个 p 顶点被一条边连到一个 l 顶点上,则说明该进程可能被指派到 l 节点。指派图的边包含有代表特定调度算法目标的某种形式的权值。为了将进程指派到特定节点,每个节点顶点都必须从图中分割出去,图的分割通过断开图的边的方法进行。为了将一个顶点从图中分割出去,必须切断开一组边,以使被分割的顶点从图的其他部分中分离出来。这样当一个顶点分割出去后,包含此顶点的子图就不再有任意边连接到原图其余部分了。例如图 7-5 描述了

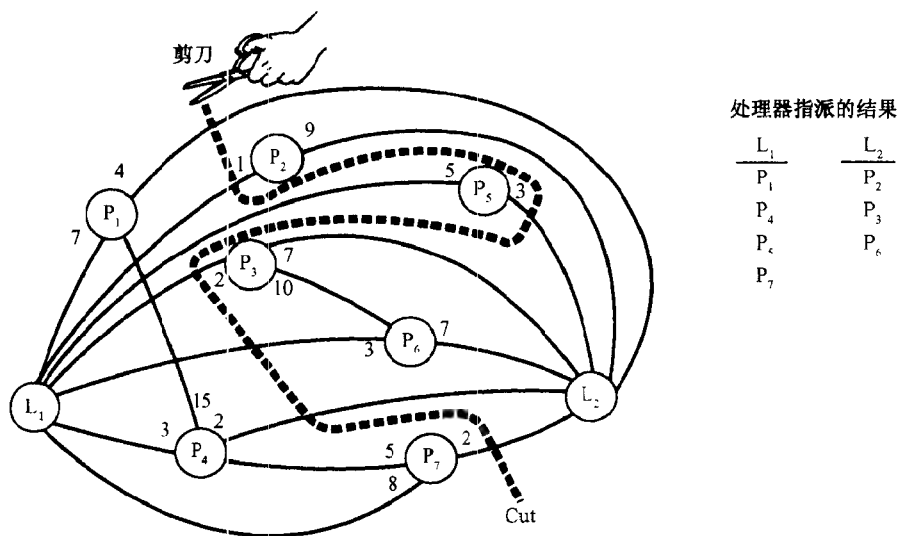


图 7-5 两个处理器的最小分割示例

一个有两个处理器的最小分割。为了移走顶点而被切断的边的集合称为分割集。每个分割集包含有一个代表节点的 l 顶点。当一个 p 顶点由一条边连到一个 l 顶点上时, p 顶点代表的进程就指派给 l 节点。分割集的开销就是为移走顶点而被切断的边的权值的总和。最小分割集表示分割特定顶点的最小可能开销。任何在分割节点的最小分割图中的任务会被考虑指派给该分割节点。处理器指派的总共开销是分割每个节点顶点的开销总和, 因此最小分割的目标就是使指派开销的总和最小。对于给定的指派图分割所有节点顶点的最小开销——最小分割代表了最优的调度方案。

我们现在讨论基于计算图的最大流量的第二个图论方法。该方法利用了**最大流量指派图**。该图的顶点如果记为 p 则代表一个进程, 若记为 l 则代表一个节点。在这种情况下, 图的边是有向的 (由箭头来表示), 它代表进程对箭头指向的节点顶点的可能的指派。每条边包含有代表该边流量的权值, 在本应用中边的权值越大说明在给定进程和节点间的流量越大。最大流量指派图的目标是将进程指派给使得整张图的流量之和最大的节点, 这通过选择带有最大权值的边来获得。在图论领域, 获取最小分割集的目标和获取最大流量的目标是等价的, 有很多算法是用这两种方法来评估一个图。选择使用哪种图论方法来解决调度问题就由应用者自己来决定。

我们现在介绍一个基于图的最小分割调度算法的例子, 这个例子所讲的是负荷平衡、全局、次优、近似、集中式的动态调度算法 (见 Lo [88])。该算法可用于一次性的指派或用于适应性的指派, 以下是性能评估的基础:

- ◆ 由进程指派所引起的总共的执行和通信开销最小化。
- ◆ 总共的干扰开销最小化, 使并发度增加。

两个进程之间的干扰开销反映了它们的不兼容性, 用来将高干扰开销的进程调度到不同的节点上。一个高干扰开销的例子是两个进程都需要大量 CPU 集中的操作, 而一个高输入/输出进程和一个需要大量 CPU 集中操作的进程就只有相对较小的干扰开销。这种算法假定编译器或系统为所有进程提供了执行和通信开销, 实际上这些值可以是近似值, 因为开销的

计算将极有可能对系统性能产生影响。这种调度算法不考虑与内存需求、最后期限及优先关系等相关的问题,因此它是一个近似算法,因为在算法后面的步骤中利用了直觉。忽略某些信息是允许的,忽略一些变量和假定它们是常量来研究和评价一个特定方法也是很常见的。

对于该调度算法,目标是整个分布式系统中所有与指派有关的执行和通信开销最小化。利用指派图,Lo的次优启发式算法由以下三个阶段组成:

1. 抢夺 (Grab): 对于每个节点,节点抢夺由大的权值表示的对它很有吸引力的所有的进程。由于这种权值很大,它代表了我们不太希望分割的边。如果本阶段结束时产生了每个节点的完全的分割,则说明我们已经得到了一个最优的调度方案,否则继续进行下一阶段。

2. 聚集 (Lump): 聚集是将所有未指派的任务尽力地指派给一个节点,即将它们聚集起来。这种可能的指派开销将进行计算并跟相应的通信开销比较,这些通信开销只有在进程没有聚集到一起时才会产生。如果通信开销比聚集开销高则算法就进行相应的聚集,否则聚集就不是最小的开销和最有效的指派,这样算法就进行第3阶段。

3. 贪婪 (Greedy): 贪婪算法的中心是确定进程间通信开销很高的进程组。这些进程需要调度到同一节点,而不同进程组可以指派到不同的节点。本阶段考虑与分割有关的开销,即通信开销和执行开销。由于并非每种可能性都被检验,所以该阶段的结果是次优的。

在 [Lo88] 中,也有在第3阶段即贪婪阶段使用更复杂标准的几个变种调度算法,但是刚才介绍的简单的贪婪方案在仿真测试中的性能几乎与复杂方案一样。

7.2.3 探查

消息可以发送给系统中的成员,以定位一个合适处理器来调度进程,这种消息称为探查。利用探查消息的调度算法可以是最优的或是次优的。最优方案的实现必须探查每个节点并分析得到的数据,而次优实现可以有选择地不去探查所有节点和/或有选择地分析从探查消息中接收到的完整信息。如果每个节点都能发送探查消息,则算法就是分布式调度算法。反之,集中式服务器是分布式系统中惟一的发送探查消息和维护这种信息的角色,则算法就是集中式调度算法。此外探查消息可用来安放进程,或用来评估重定位的可能性,这样就是以适应性的方式执行调度。最后,探查消息可以被欠载节点用来寻找更多的工作。

一个使用探查消息的调度者例子见 [ELZ86],具体地说它是一个负荷分担、全局、次优、启发式的静态调度算法。在该方法中,认为进程的定位开销比通信开销更重要。定位开销是将进程调度到与原先所处系统不同类型的系统有关的开销。更为看重定位开销的理由是因为涉及到异构系统,即在不同类型系统上执行一个进程的有关开销。当然如果进程代码不能移植到特定节点中的特定宿主系统上,则该节点就根本不予考虑。

该调度算法的策略涉及到利用探查消息。当某节点需要远程调度或迁移一个进程时它初始化三到五个探查消息(测试表明更多的探查消息不能显著地提高性能)。被探查的节点必须是与发送探查消息的系统同构的(用权值来影响)。在同构系统可用时,接收探查消息的节点是随机选择的。这些探查器测试节点的探查限制,代表了若进程迁移到该节点时节点将处在什么样的负荷状况(见第2章所述)。如果探查的节点限制表明节点处在过载状态,则该节点就会被排除在外。但是若探查的节点限制表明节点处在欠载状态,则立即进行调度将进程迁移到该节点。注意被选择节点的负荷状况在发送探查消息时刻和进程迁移到达时刻之间可能会发生变化,这不会也不能影响进程迁移决定。而且探查没有进行寻找和确定最佳节

点的动作。此外探查只能确定节点整个的负荷状况而不能计算负荷状况的具体值或索引。

7.2.4 调度队列

集中式操作系统很久以前就开始使用调度队列，所以在分布式环境中使用该方法就不奇怪了。当使用调度队列时，分布式调度程序一般维护着分开的全局和本地队列。本地队列是存在于集中式系统中的标准队列，维护着在本地运行的进程列表。而全局队列是用于非指定本地调度，而是预计可能运行在整个分布式环境中的进程。全局队列是否允许进程确认执行环境要求，例如希望的平台或机器，则依特定的实现而定。虽然每个节点可以拥有全局队列的本地拷贝，但一般有一个集中式服务器用来保存和更新全局队列。使用调度队列的一个优点是使用调度队列的调度算法可以很容易地结合到现存操作系统的上层。分布式调度程序可以简单地调用一个将被调度的进程。当调度该进程时，它执行分布式调度算法，将全局进程带到本地处理器供调度。在没有基层操作系统时，调度队列能很容易地实现分布式调度。全局与本地调度队列将如何由系统处理依赖于具体操作系统的策略。

使用调度队列的调度算法的一个例子见 [Bla90]。该算法涉及到 Mach 中使用的处理器分配调度。Mach 是由卡内基-梅隆大学开发的用于研究的分布式操作系统。Mach 已由开放软件基金组织 (OSF) 所接管，该操作系统的大部分已结合到了 OSF/1 操作系统中。这种调度算法是可以用于负荷平衡或负荷分担的强制、次优、启发式、适应性、分布式的动态调度算法。

调度在线程级进行，每个节点都有一组本地运行队列和一组全局运行队列。每个单独的运行队列表示了不同的优先级，需要被调度的最高优先级的线程的位置用一个 hint 变量来表示，见图 7-6 所示。这样如果 0 级的到 4 级的优先级的运行队列为空，那么 hint 变量的值将为 5，表示的是包含一个线程被调度的最高优先级的运行队列。对于调度目的，最高优先级的本地进程由调度来执行。线程优先级由基本优先级和代表最近处理器利用率的补偿值组成，补偿值也反映了系统的当前的负荷状况。也有一种机制基于线程的年龄来增大线程的优先级。每个运行队列包含有一个计数器用于它所维护的线程数目和互斥锁数目。全局运行队列只有在由队列的计数器表示的本地运行队列为空时才会被查找。如果没有本地运行队列，那么有最高优先级的全局线程就会被调度执行。本地运行队列用于限制在特定处理器上运行的线程，如果一个线程设计成用在特定体系结构的处理器上，那么它就可能是那种受限制的线程。Mach 共使用 32 个运行队列，最小数字的运行队列代表最高的优先级。

Mach 调度允许使用由用户提供的 hint 变量。具体地说，用户可以表达不提倡提示和阻止提示。阻止提示使得用户可以进行并发控制而提高系统性能。阻止提示有三个等级：温和

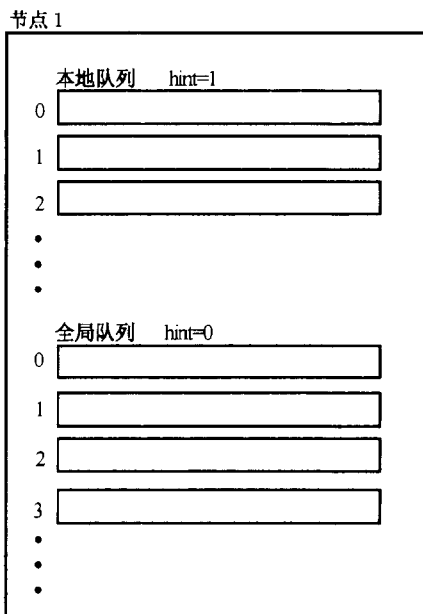


图 7-6 带有运行队列和 hint 变量的节点

级、强烈级和绝对级。温和提示建议不要调度一个线程或将线程切换出去。而强烈提示则会暂时地减小特定线程的优先级，这通过让可能潜在的拥有锁的线程执行以希望它释放出锁住的资源来增加并发控制的性能。绝对提示禁止一个线程在限定的时间内被调度，这给低优先级的线程提供了执行的机会。如果低优先级的线程拥有一个高优先级线程需要的锁，这种方法就可以让低优先级的线程执行然后释放锁。

传递提示允许线程告诉调度程序执行一个特定的线程而不是提供传递提示的线程。该新线程马上就获得对处理器的控制，这样就旁路了运行队列。这种传递可以由原先的线程显式地取消或最终超时。传递技术也在低优先级线程控制了高优先级线程需要的并发控制锁时很有用。在这种情况下，高优先级线程允许将处理器传递给低优先级线程。当并发控制锁被释放后，高优先级线程可以显式地取消传递并继续进行计算，而先前被锁住的资源这时就可用了。

为了允许集体调度，Mach 提出了一个称为处理器集的抽象概念。如果用户有一组要求集体调度的线程，就可以将它们指派给特定的处理器集。一旦将某个线程指派给一个处理器集，它就必须在这个处理器集中执行，而且处理器集只能运行指派给它的线程。当内核试图调度已指派给某处理器集的线程时，它利用这种信息来分配特定处理器到处理器集以保证所有有相关线程同时调度。

165

7.2.5 随机学习

随机学习 (Stochastic Learning) 是基于以前的行为来确定当前的最佳动作的启发式方法，即从经验中学习。所有的调度行为与一个概率关联。分布式系统建立时，所有的概率都初始化为相同的值，这些概率用来选择一个节点。在作出调度选择并且进程被送到调度的节点后，目的地节点提供反馈信息给源节点。根据对调度选择的评估该信息可以奖励点数或罚减点数。如果目的节点过载，则它就会发送消息给罚点，若目的节点欠载它将发送消息给奖励点数。这些点数可以用来为刚才的行为相应地调整概率，系统就是这样基于奖励/处罚反馈来进行学习。

这种方法可以进一步扩展以允许调度程序记录系统状态和按照系统的各种状态来行动。整个系统的每个可能状态都由一个表项表示成一个**自动机向量**。每个索引代表了分布式系统的不同工作负荷。当使用调度算法进行调度时，向量的内容同与节点相关的概率一起被使用。对自动机向量的使用，要求节点周期性地报告它们的负荷状态给分布式系统中的各个成员。这种信息用来计算系统的状态，保证了当进行调度决策时引用的是正确的自动机表项。这种方法可以利用集中式服务器来维护自动机向量和收集到的工作负荷信息，但它也适合于分布式的实现。

我们现在讨论一个如 [Kun91] 所述的使用随机学习自动机的调度算法的例子，该文给出了一个强制、次优、启发式、一次性的、分布式动态调度算法。这种算法当本地处理器处在欠载状态时不允许进行远程调度，而且如果分布式系统中的所有节点都处在过载状态时也不允许进行远程调度。基于从节点中收到的活动消息计算一个系统范围的状态，活动消息被节点用来传送一个过载或欠载状态。节点的工作负荷状态是依据当前被指派给节点的进程的处理时间要求和它们的输入/输出量及内存需求而定。该算法已通过以下的工作负荷指标的方法进行测试：

- ◆ 平均一分钟的负荷。
- ◆ 剩余内存空间。
- ◆ CPU 的空闲时间量。
- ◆ 上下文切换的频率。
- ◆ 系统调用的频率。
- ◆ 运行队列的长度。

166

工作负荷指标按效果增加的顺序给出。这样节点使用平均一分钟的负荷指标时效果最差，而使用运行队列长度指标时效果最好。Kuntz 发现两个指标之间有 32% 的响应时间的差别。另外也使用工作负荷指标组合的方法测试过，但实验结果未反映出系统响应时间的减少。

基于工作负荷的状态信息每 8 秒钟就给系统中的节点发送一次，系统状态就计算和映射到对应特定自动机向量表项的自动机状态中，这个表项以后就用到调度决策中。当调度程序初始化时所有的概率是相同的，这对每个远程节点被选择出来执行远程进程给予了相同的可能性。概率依据从选中节点反馈的奖励或处罚消息来进行调整，这种概率以后就用于随机选择将进程调度到那个远程节点。

7.3 协调者选举

在分布式计算的每个与集中式解决方案有关的问题中，有一个集中式服务器在系统中扮演着关键的角色。在本节里，我们介绍用来确定分布式系统中的哪个特定系统充当集中式服务器的一般方法，而且该方法还可以在当主服务器出故障时用来指定一个新的服务器。引用得最多的协调者选举算法是由 Garcia - Molina 提出的 [Gar82]，那篇文章将选举算法称为“欺负算法 (Bully Algorithm)”，该算法同所有在文献中给出的调度算法一样需要使用参与者的地址号。当一个节点注意到没有服务器时，节点发送消息到系统中的所有成员要求成为协调者，这种消息包含有节点的地址号。如果分布式系统中大号地址的任何成员希望成为协调者，则大号地址的节点可以“欺负”小号地址的节点而自动地成为协调者。当然欺负者也会被分布式系统中更大号地址的另一个成员欺负，见图 7-7。这样当形势紧急时大号地址的节点只要愿意就总是能够成为协调者。如果没有节点去夺走小号地址节点的协调者地位，则该小号地址的节点就成为协调者。协调者欺负算法的例子见详细说明 7.3。

167

详细说明 7.3 一个欺负协调者算法

对协调者通告的反应

```
IF receive(I_want_to_be_coordinator, recipient_address)
    & recipient_address < my_address
    & I_want_to_be_coordinator
    THEN announce_desire_to_be_coordinator
        & send(I_want_to_be_coordinator, my_address)
```

宣告渴望成为协调者

```

IF no_response_from_coordinator
THEN forall(participant_address)
    send(I_want_to_be_coordinator, my_address)&
    settimer

```

等待应答

```

IF receive_message & recipient_address > my_address
THEN cannot_be_coordinator
ELSE IF timer_expired
    THEN I_am_coordinator

```

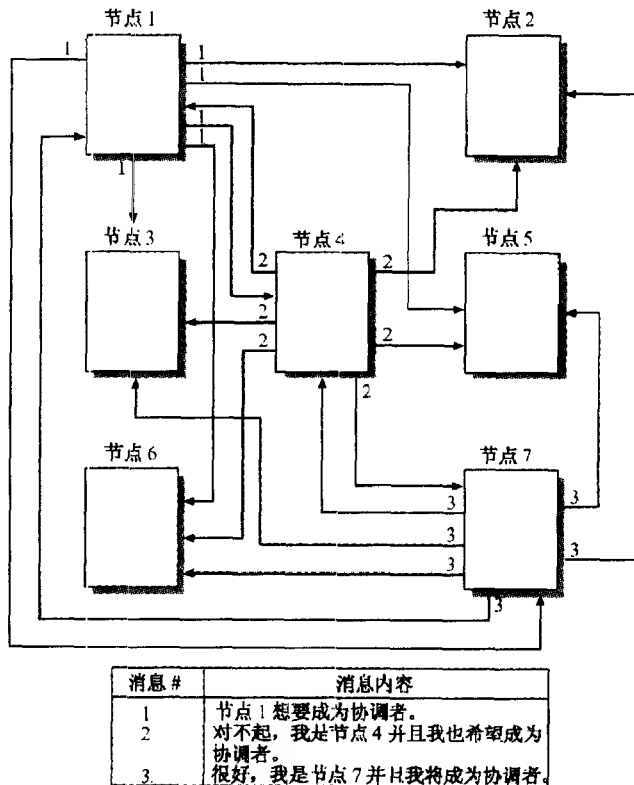


图 7-7 欺负算法

这种同样的过程不但可以在协调者失效时进行, 而且在选举新协调者时也可以进行。该算法的一种变种是只发送消息给地址号比自己大的节点, 而不是所有节点, 因为小号地址的节点不能阻止特定成员成为协调者。节点的地址在网络安装时就已确定, 并且一般对于系统来说是不可改变的, 这样就不允许节点改变它的地址来“插队”。总之地址号大就能允许特定分布式系统成员统治系统并每次都成为协调者。

7.4 孤儿进程

分布式环境中的一个重要问题是对**孤儿进程**（Orphan Process）的处理，这个问题最初是在 [Nel81] 中提及。在当父进程过早地终止时就会生成孤儿进程，对于远程过程调用这个问题就更常见。

任何由终止进程创建的进程都是子进程，它将继续执行下去，除非分布式系统的进程管理中包含某种处理孤儿进程的机制。孤儿进程是没有人接受其返回值的进程。而且，子进程甚至也可以创建自己的子进程，这样就将问题进一步复杂化了。当一个进程在进行没有父进程的执行时，它会浪费资源和破坏数据。浪费的资源最常见为 CPU 周期，还有网络及存储资源。此外孤儿进程会以不可逆的方式修改共享数据。如果子进程控制了某种类型的并发控制锁，则当告诉它终止执行时，它就必须释放所有锁，但是这不能保证数据不被不可逆地破坏。在前面学习调度时我们知道进程可以被调度到整个分布式系统中的任何地方，这样这些孤儿进程可以在整个分布式系统中的任何地方执行。我们以下介绍管理及处理孤儿进程的三种方案。

7.4.1 孤儿进程清除

孤儿进程清除要求当进程在系统或硬件崩溃后苏醒过来时清除它所创建的子进程。为了完成这个操作，进程必须保存所有它所创建的进程的清单。父进程可以要求它的所有的子进

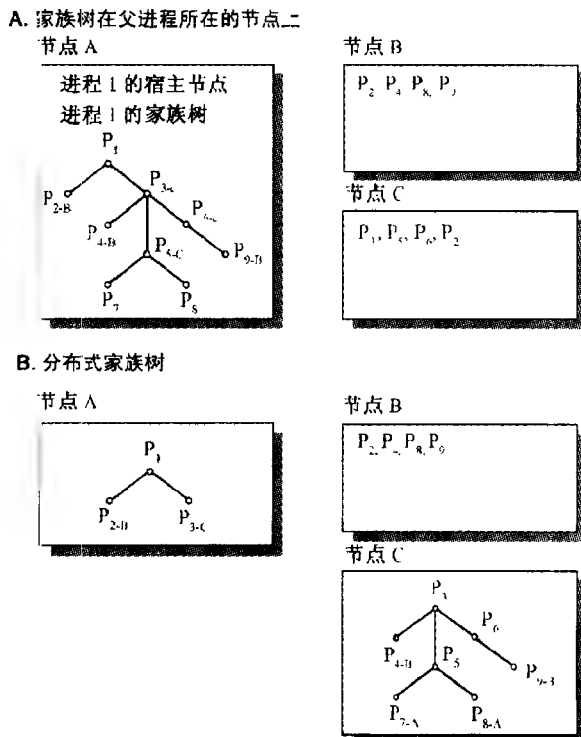


图 7-8 孤儿清除的家族树

程在创建子进程时通知父进程或要求子进程保存自己的子进程清单,如图 7-8 所示。在任何一种情形下,这种家族树都必须保存在稳定存储器中。当系统恢复后,它检查家族树以确定在它崩溃时刻正在执行的进程。这时清除操作必须开始,因为家族树的所有成员都被通知了它们的父进程已经死亡并停止执行。如果要求子进程维护自己的家族树,则它必须通知它所有的子进程终止执行。

这种方法的最大的缺点之一是它给父进程所在的节点带来了巨大的负担。这种负担不

详细说明 7.4 孤儿清除算法

在家族树中记录入口

```
Repeat forever
{
  Foreach process (PID)
    //PID 是进程标识符
    // 对于每个进程都必须执行这个操作
    IF system created (PID)
      // 新创建的进程, 系统是其父进程
      THEN
      {
        Create family tree (PID);
        Flush(family tree, stable storage)
        // 必须创建新的家族树并存放在稳定存储器中

        add (PID, process table);
        // 添加该进程到活动进程列表中
      }
      ELSE
      {
        // 这不是一个系统创建的进程,
        // 而是另一个进程的子进程
        parent = look up parent (PID)
        add to tree (parent, PID)
        // 将新进程加入到它的父进程的家族树中
      }
      //ELSE 结束, 需要添加到父进程的家族树中
    }
  }
  //向家族树中记录结束
```

崩溃后恢复

```
Foreach process in process table
{
  foreach process in tree (PID)
    send clean up message;
    // 通知所有子进程终止运行
}
```

但包括家族树必要的存储开销，而且还包括不断更新家族树和维持家族树的准确性所需要的开销。孤儿进程清除算法的一个例子见详细说明 7.4。

7.4.2 子进程限额

管理孤儿进程的第二种方法要求所有的子进程得到称为**子进程限额**的时间限额，这种限额给子进程提供了有限数量的时间来运行。当限额用完时，子进程必须从父进程那里请求另一个限额，如果父进程已经终止，则子进程就不能获得限额，这样它就不能继续执行，如图 7-9 所示。若父进程仍然是活动的，则父进程将会给该子进程分配一个时间限额。同样地子进程可以创建自己的子进程，这些第二代和可能的第三代子进程可以从它的父进程或继父进程中获取限额。当子进程提供限额给它自己的子进程时，它不可以提供超过自己剩下的限额加上少量的机动时间之和。这少量的机动时间使得直系父进程在接收到它的子进程的请求之前可以获取它自己的限额。未获得限额的子进程不能提供限额给它自己的子进程。这样，当一个父进程过早终止时，它的所有后代进程由于缺乏限额而最终将会终止。

虽然这种方法不需要花费父进程所在系统的大量开销，但是它导致了额外的网络流量。而且它会干涉子进程的执行，子进程需要不时地停下来请求限额。此外当系统从崩溃状态中进行恢复

时，它不能马上就开始执行进程。如果它马上开始执行，崩溃前的子进程可能会请求限额，这样父进程就不能将它的子进程与在系统崩溃前就成为孤儿的进程区分开来。因此系统必须在崩溃后开始的任何进程执行之前等待一个限额周期。

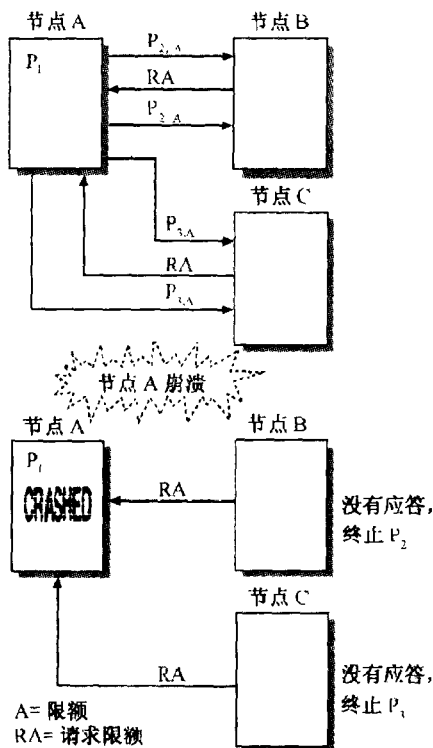


图 7-9 子进程限额

7.4.3 进程版本号

进程版本号可用来区分一个已崩溃进程的子进程和进程在崩溃后产生的子进程，这种方法需要每个进程保存一个父进程的版本号。当系统崩溃时，它宣告一个新的版本号给整个分布式系统，这样就有两种实现方案可用于处理由该系统创建的子进程：

1. 可以强迫子进程终止运行，因为它有着过时的版本号，见图 7-10 所示。
2. 子进程可以首先定位它的父进程（从它创建后可能会迁移），如果不能定位父进程，则子进程必须终止运行。

在这两种情况下，该方法在每次系统失效时都需要进行大量的通信，而且和以前崩溃的系统没有任何联系的系统也必须接收和处理宣告新版本号的消息。

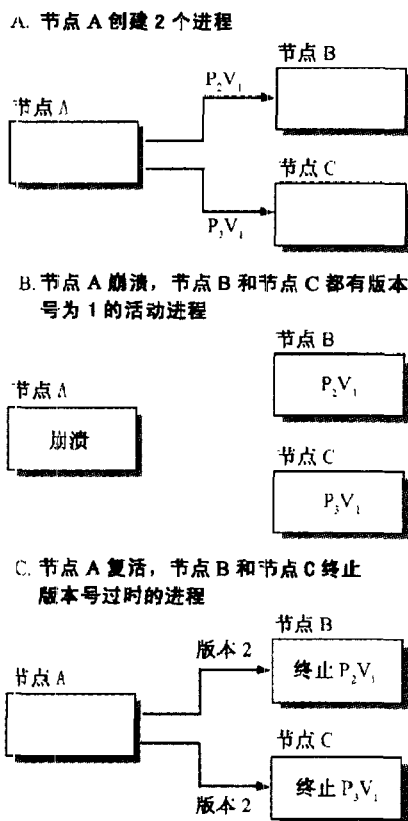


图 7-10 进程版本号

7.5 小结

在本章中, 我们深入讨论了分布式进程管理的各个方面。具体地说, 我们讨论了分布式进程调度所涉及的问题, 集中式算法当中选举协调者进程, 以及处理孤儿进程所涉及的问题。当设计一个控制进程指派到远程节点的分布式进程调度算法时, 有多种选择方案可用。由于选择的多样性, 可以有 42 种不同类型的调度算法。一些算法很自然地用于多种调度算法类型而另一些则不能。具体地说, 虽然最优化解法由于执行起来速度慢而一般不采用, 但是所有能用于一次性调度算法的算法均可以用于适应性调度。一些算法就调度方案绑定时间而言并不特别, 而另一些, 例如探查算法利用了进程绑定的系统类型, 这样在确定可以执行远程节点时能够减少节点搜索空间。所有调度算法使用的方法都可用于负荷平衡和负荷分担, 而且可以使用这些方法的组合。例如随机学习可以很容易地改为同探查算法一起作用, 而代替由每个节点广播它的负荷状态。可以将图论方法应用于随机学习的初始调度信息。使用点数可以用来确定进程的优先级并同调度队列结合在一起使用。尽管对于调度已经进行了大量的研究并有许多可用的信息, 但目前它仍然是一个非常活跃的研究领域。一些比较困难的问题涉及到工作负荷描述的定义和它们相应的反映节点负荷状况的阈值。另外的一些研究工作主要是改进调度算法的效率。

关于协调者选举, 通用的规则是资深惯例。资历以节点的地址来确立, 地址号越大的节

点其资历就越老。如果一个节点有最大的地址号并且它想做额外的工作，则协调者选举算法通常就允许该节点成为协调者。另一方面，处理孤儿进程不是一件容易的事情，每种处理方法都有缺点，但是这些缺点都未超过不使用孤儿进程处理方法所带来的损害。当我们讨论各种用于终止进程的方法时，都没有论及这些终止方法将会如何影响数据的问题，这种问题可以用一个词来回答：事务（这个主题在第9章讨论）。

最后，要强调的是分布式进程管理仍然是一个活跃的研究领域，而且还没有特别好的解决方案出现。除了在本章讨论的主题之外，还有以下领域仍然需要重点研究：

- ◆ 怎样才能最好地解决处理器拥塞。
- ◆ 怎样处理、确定和防止局部系统过载。
- ◆ 怎样确定理想的进程级粒度和用有效的算法来处理这个问题（大量的小进程分布处理会比分布式处理方法带来更多的问题）。
- ◆ 怎样确定理想级的处理器粒度。

考虑到这些相对未解决的问题以及本章介绍的这些主题，所以说分布式进程管理在实践中仍是一个困难的课题，但同时又不失为一个非常好的研究、基金资助以及学位论文题目的领域。或许解决的答案在于采用不同学科的方法，例如商业管理（分割和分布（或委托）工作负荷是管理科学的一个常见的问题）。

7.6 参考文献

文献 [BGMR96] 和 [BuScSu97, Kop97, and SSRB98] 分别为分布式调度算法和实时调度提供了更详尽的具体的算法分析信息。一些与本章内容有关的传统的研究论文包括 [ArFi89, Bla90, CaKu38, EAL95, ELZ86, Esk89, Fer89, FeZh87, FYN88, Gar82, KrLi87, Kun91, LHWS96, Lo88, MuLi87, Nel81, Ous82, PaSh88, PTS88, SHK95（一组经典论文），SKS92, VLL90, 和 WaMo85]。

在第1章列出的 Internet 搜索站点提供了 Internet 中与分布式进程管理有关的资源的最佳的起始点。

习题

- 7.1 指定负荷分布目标对负荷平衡的不利方面是什么？
- 7.2 在分布式系统中可能实现真正的负荷平衡吗？它有必要实现吗？为什么？
- 7.3 在第1章，我们介绍了用工作站模型和处理器池模型实现 C/S 环境的概念，哪种模型更有助于实现负荷平衡的目标？为什么？
- 7.4 列举三种提高调度效率的可行措施，并列举每种措施至少一条优点和一条缺点。
- 7.5 列举次优近似方案与次优启发式方案相比的一个优点，列举次优启发式方案与次优近似方案相比的一个优点。
- 7.6 在分布式系统中哪种处理器绑定时间类型更好？为什么？
- 7.7 在分布式调度算法中简单使用点和经济使用点方法相比，其优缺点各是什么？
- 7.8 为用于分布式调度的简单使用点方法写一个仿真程序：如果两个进程属于具有相等使用点的节点，则使用先到先服务原则。程序应当从一个文件中读取输入，希望的输入的格式如下：

时间 节点 进程号

时间表示事件发生的时间，节点指定引发事件的位置，进程号用来确定特定进程。如果进程号的值为0，则事件表明该节点为进程执行提供帮助，任何非0进程号都表明进程需要帮助。程序的输出应当包括一个表明何时事件发生，何时在某节点上的一个进程被调度及节点号的相关信息。此外，程序应当表明何时节点收取和支付使用点。以及每个节点的最终使用点列表。可以假定有一个集中式服务器来维护使用表。也可以假定如果一个节点需要帮助而没有节点可提供帮助则也向它收取一个使用点。用至少四个不同的测试文件来测试你的程序，注意每个测试文件的侧重点不一样。

- 7.9 本文中讨论的用于进程调度的图论方法是一种集中式解决方案，请问若将这种方法用分布式方案来实现会遇到什么困难？
- 7.10 为什么 Lo 的 [Lo83] 的调度算法是一种近似方法？说明你的理由。
- 7.11 [ELZ86] 中所述的探查算法的例子是优先考虑定位开销，它的优点是什么？其可能的缺点又是哪些？
- 7.12 当在分布式调度中利用本地和全局调度队列时，为什么首先调度本地队列？其优缺点各是什么？将提示变量 hint 结合到使用队列的分布式调度算法的优点是什么？
- 7.13 请阐述 Mach 调度允许使用用户提示的潜在优点，并给出一种可行的替代方案。
- 7.14 请阐述在随机学习的调度算法中使用自动机向量的优点，不使用自动机向量的方法的缺点是什么？
- 7.15 指定一个过载的节点为协调者会对分布式系统的性能带来损害，欺负算法是怎样选择合适的系统成为协调者的？
- 7.16 请阐述处理孤儿进程的三种方法的优缺点。

176

177

第 8 章 分布式文件系统

在操作系统中，文件系统是非常重要的子系统。该子系统提供了辅助存储器的抽象视图并负责全局命名、文件访问和全面的文件组织。这些功能分别由名字服务（8.1 节）、文件管理器服务（8.2 节）及目录服务（8.3 节）来提供。虽然分布式文件系统的实现必须提供这些服务，但是它们不需要在不同的服务器中实现。除了分布式文件系统需要提供的服务外，本章还介绍两个著名的分布式文件系统实现的实例，它们是 8.4 节中的网络文件系统（NFS）和 8.5 节中的 X.500 规范。

8.1 分布式名字服务

分布式名字服务主要是涉及与文件名有关的问题，一个常见的问题是“文件名是什么？”，分布式文件系统会反问“你打算让文件名传达什么信息？”。文件名可以传达文件类型及文件所在位置信息。我们将在 8.1.1 节讨论文件的类型，8.1.2 节介绍位置透明性及实现位置透明性的必要的要求，全局命名和命名透明性在 8.1.3 节讨论。

8.1.1 文件类型

各种操作系统允许不同类型的文件和以不同方式支持文件系统。在文件名中，文件类型一般通过文件名中“.”后的部分——扩展名来表达。准确的扩展名通常认为是一种惯例而非规定，尽管某些应用需要通过它们来识别文件。以下列出了部分常见的文件类型及可能的扩展名：

- ◆ 库文件：可在用户程序中使用的通用例程，这种文件使用扩展名 *lib* 和 *dll* 等。
- ◆ 程序文件：用户所写的程序。这种文件使用扩展名来表明所使用的特定程序语言，例如 *c*，*cpp*，*p* 和 *pas*。
- ◆ 目标代码文件：未链接的、已编译过的、通常以机器语言表示的程序，这种文件使用的扩展名有 *o* 和 *obj*。
- ◆ 压缩文件：为了节省存储空间而进行压缩的文件，这种文件使用扩展名 *Z*，*zip* 和 *gz* 等。
- ◆ 存档文件：相关的文件为了便于存储而组合成的单个文件，这种文件使用扩展名 *tar* 或 *arc* 等。
- ◆ 图形图像文件：用于打印和浏览的二进制或 ASCII 文件，这些文件常用的扩展名有 *dvi*，*ps*，*gif*，和 *jpeg* 等。
- ◆ 声音文件：保存声音数据的二进制文件，这种文件使用扩展名 *midi* 和 *wav* 等。
- ◆ 索引文件：索引文件通常包含其他主机的索引信息，这种文件使用扩展名 *idx*。
- ◆ 文档文件：由字处理器创建的文件或由排版系统转换成的文件，这种文件常用的扩展名有 *doc*，*wp*，*tex* 等。

名字服务必须允许所有可能的文件类型。某些操作系统为不同的文件类型提供支持。例如某些操作系统能够识别二进制文件，但不允许用户浏览上述类型的文件。而其他操作系统

为一个给定的文件类型调用适当的应用程序。某些编译器不编译未以正确的文件名结尾的程序。在 UNIX 中有两种另外的、不是通过扩展名来区分的文件类型：文件名以“.”文件开头的不可见文件和目录。当系统自动地调用应用程序时，如果有可能会损害系统的灵活性，从而会损害用户的灵活性时，文件类型支持是很有用的。具体地说，如果本来就支持文件类型，那么每个新的文件类型必须连同它的应用程序信息一起报告给操作系统。这种全面文件的类型支持使得操作系统在支持以后出现的新文件类型时变得很困难。

180

8.1.2 位置透明

文件名能传达的第二种信息是文件的位置。如果文件的位置被传送，则名字中可以包括位置、机器和文件名，例如 myuniversity.edu: /violet/u/galli/book/chapter8。若分布式系统希望提供位置透明，那么必须通过**全局命名**（见 8.1.3 节）来提供名字透明。位置透明性依赖于名字与位置独立，这是非常重要的而不是无关紧要的。设想如果电话号码是位置独立的，那么电话号码中就再也不需要区号了，因为区号是用来表明电话号码持有人所在的位置。其优点是如果你迁到你们国家的另一个地方或迁到另一个国家时你的电话号码不用改变，所有在你搬迁之前知道你的号码的人仍然可以与你联系上而无需知道你已搬家。这显然是很方便的，特别是对于一个经常搬家的人来说。这种号码称为**统一号**（universal number），如号码 800 和 888。同样地，在一个分布式系统中文件可能会迁移到一个新的位置或一个文件在整个系统会存在多份拷贝。若没有位置透明性，用户和/或他们的系统将需要不断地跟踪每个文件的每份拷贝的位置，并显式地表示出文件的位置以进行文件访问。这将极大地损害分布式操作系统的基本思想，因为分布式操作系统的根本目标就是提供统一的系统视图，这样就可以隐藏存在多个系统和多个节点的事实。详细说明 8.1 描述了位置透明性缺点的一个实例：路由。

详细说明 8.1

位置透明性的路由问题

位置透明性的一个问题涉及到信息的路由。当名字中包含有位置信息时，这种信息可以用来帮助路由，而不需要有完整的主名到位置的转换表。我们讨论 IPv4（因特网协议版本 4）和将 IP 地址的使用与电话号码的使用相比较，作为路由在位置透明情况下遇到困难例子。IP 地址不提供任何位置的信息而电话号码则提供位置信息。注意到任何使用文件名描述及 IP 地址的命名系统都不提供位置透明性是很重要的。这种文件名依赖 IP 地址位置，如果它迁移到由 IP 地址来标识的另一个节点，则文件名需要改变。此外，如果特定节点显示或创建一个跟它的 IP 地址数相关的位置，则该节点就消除了路由的位置透明性特点。若节点位置改变，则路由系统就会引发异常消息或要求重新指定一个地址号。对于这个例子，我们只讨论 IP 地址（没有层次结构的）同非统一电话号码的比较。

我们首先介绍提供位置信息的电话号码是怎样对路由提供帮助的。电话号码提供了以下三级位置信息：（1）国家代码；（2）区域代码（区号）；（3）本地交换码。前两级号码是可选的，并且如果没有被包括进来就假定目的端与源端有相同的前两级号码值。电话交换机检查国家代码和区域代码以将路由呼叫到大概正确的物理位置。一旦到了正确

的大概位置，则基于本地交换码特定的位置将进一步具体化。一旦将呼叫交换到了恰当的本地交换局，那么基于最后的几位数字呼叫就可交换到确切的位置。如果接收到一个不是对本地位置的呼叫，则交换机只是基于以下规则将呼叫简单地切换到适当的位置：

1. 如果呼叫的目的地是在我的区域号码内但交换局不同，则将呼叫切换到相应的交换局。
2. 如果呼叫的目的地的区域号码同我的区域号码不同，但是属于我国的一个区域，则将呼叫切换到相应的区域中。
3. 如果呼叫目的地是外国，则将呼叫切换到相应的国家。

在相应层次上接收到呼叫时，就按照需要将呼叫交换到相应的区域代码和/或交换局中，如图 8-1 所示。

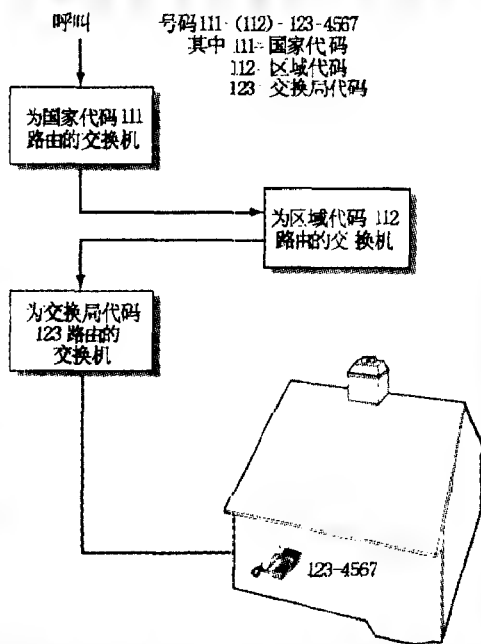


图 8-1 电话路由

相反，IPv4 路由器不具有上述的优点。IP 地址被分成不同的部分，为了不让本例被无关的细节复杂化，地址的一部分是确定一组计算机的位置，而剩下的部分则确定该组计算机中特定计算机的位置。一旦某组机器的位置已经知道，则将消息发送到该组机器，该组机器的交换机接着就将消息路由到特定的机器。但是组地址并未揭示任何位置信息，如果目的地位置不为本地路由器所知道，则一个位置信息请求就必须传送到一个知道更多位置的称为网关的节点。网关是拥有非常大的到位置转换表地址的节点，转换表中有目的机器的位置信息。一旦获得了这种信息，路由的正确的路径就会被选择。由于可能有超过二百万个的机器组，这种完整的表非常大。一种对 IP 地址的帮助方法是使用符合域名系统规范的符号名字，见详细说明 8.2。

8.1.3 全局命名与名字透明

一个支持位置透明的分布式文件系统必须支持全局命名。但是，全局命名本身并不意味着位置透明。具体地说，全局命名分配每个文件（或对象）一个全局惟一的名称。如果该全局惟一名称是位置/机器名/文件名，则系统就不支持位置透明。对于一个支持位置透明的系统，它必须也同时支持名字透明。名字透明也需要全局惟一的名称，但这种名称不能以任何方式包含位置的任何信息。有时会错误地认为名称可以同服务器相关联，并且仍然能提供名字透明。但是如果服务器名称包含在文件名中，那么文件就要求一直保留在那个特定的服务器上。这限制了整个系统的灵活性并妨碍了位置透明，因为这种安排将禁止文件迁移到另外的服务器上。

全局名字空间需要以下的一种或两种类型的解析：

- ◆ 名字解析。
- ◆ 位置解析。

名字解析是将用户友好的符号文件名映射到计算机文件名。符号文件名习惯上涉及到字符，而计算机文件名通常（不是非常用户友好的）是数字。虽然数字可以更加容易地被计算机处理，但这种方法在人类用户中并不流行^①。这样每个全局惟一数字名称必须映射到一个全局惟一的基于字符的名称。在第6章中我们学习过 Clouds 和 Amoeba 是怎样实现全局命名的，具体地说两个系统都提供了（二进制）数字的全局惟一名称，这些非用户友好的名称自动地映射成了用户友好的（ASCII）名称。虽然 IPv4 没有涉及到文件名解析，但它为全局名字解析实现提供了一个极好的实例，见详细说明 8.2。

详细说明 8.2

IPv4 名字解析

IPv4 名字解析由域名服务器（DNS）来执行，这个系统用来将 IP 地址解析到符号名称，它基于层次符号名称空间，如图 8-2 所示。每个符号名都包含有一个域名，几个域名的例子如下：

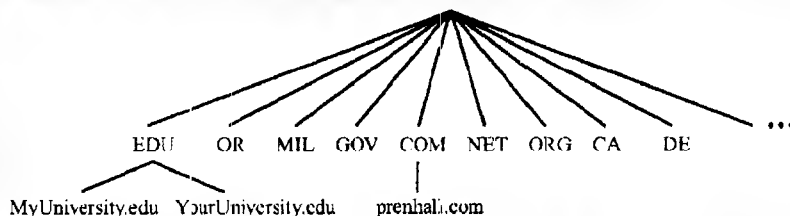


图 8-2 IP 层次性名字空间

- ◆ EDU：教育机构
- ◆ ORG：非赢利性组织

① 最早版本的 IP 只允许数字 IP 地址，但它不很方便，于是符号名称就出现了。

- ◆ MIL: 美国的军事机构
- ◆ GOV: 美国的联邦政府机构, 例如 NSF (国家科学基金会)
- ◆ COM: 商业组织
- ◆ NET: 互连网服务提供者
- ◆ 国家代码, 例如 CA (加拿大), DE (德国), NE (荷兰) 等。

注意到美国的域名不需要有国家代码, 这和英国的邮票不需要印 UK 的原因是一样的: 这是由她们首创的。换句话说来说, 由于我们所知道的 Internet 是从美国发展起来的, 这样所有美国的域名就不需指定国家, 但所有其他的域名需要指定两个字母的国家域名代码。本地节点首先尝试进行名字解析, 如果本地没有目的地位置的信息, 则消息就路由到该域相应的名字解析服务器中。整个域名服务器组织成树形结构, 每个域代表一个分支。每个节点代表一个完整的域名, 且这个特定符号名对应着特定的 IP 地址。

位置解析涉及将全局名字映射到位置。若名字透明和位置透明都被支持的话, 这是一个很困难的任务。详细说明 8.1 说明了与位置透明有关的路由困难。我们现在介绍当支持上述的透明性时, 映射名字到位置的各种方案。位置解析可以由使用集中式服务器的集中式方案来实现或由分布式方案来实现。同所有的集中式解决方案一样, 在位置解析中使用集中式服务器将产生关键部件和一个系统瓶颈。分布式解决方案涉及到所有位置都维护完整的位置解析表, 如果分布式系统非常大的话那么这种方法是行不通的。因此任何大型分布式系统都需要用带有多个位置解析服务器的分布式解决方案, 每个服务器负责所有名字集合的特定子集, 如图 8-3 所示。用一个服务器位置映射表来查询, 以确定系统中哪个服务器负责哪组名字集合的解析。将这些名字分割到各个服务器中有两种主要的方法。第一种方法是应用某种

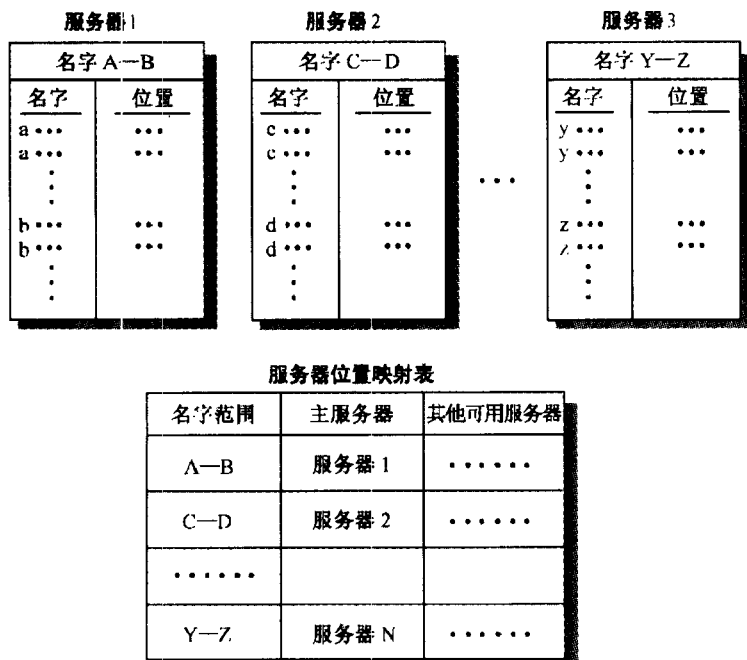


图 8-3 名字解析的分布式解决方案

散列函数到名字中, 这种散列函数的值表明哪个服务器负责维护该位置信息。第二种方法是基于文件类型来确定位置解析服务器, 每种类型的文件由不同的服务器来提供位置解析服务。无论是采用哪种方法, 当文件重定位时, 负责其名字解析的服务器信息集就会更新, 这种信息集可以用表或树形结构的方式来存储。详细说明 8.3 提供了一个位置解析的算法例子。分布式解决方案通常允许服务器复制, 即可以有多个服务器同时负责同一组名字子集的解析, 与复制有关的问题将在下一节讨论。

详细说明 8.3 分布式位置解析算法

```
location function location_resolution(name)
    // 返回文件的位置
    // 输入为文件名
    // 一个可能的实现
{
    IF local_request(name)
        // 确定文件是否在本地保存,
    THEN return_location(self)
        // 如果是则返回保存的地址
    ELSE{
        address_code = distributed_addresses(name);

        // 为要求的文件名执行映射函数以确定相应的地址码
        // 这可以是一个散列函数或某种基于文件类型的函数

        address_holder = location_holder(address_code);

        // server_location 是将代码转换到
        // 位置的表或树, 如果有多个服务器维护
        // 该信息, 则选择其中一个服务器并将该
        // 服务器名放到变量 address_holder 中

        location =
        request_for_address(address_holder, name);
        // 请求特定文件服务器返回
        // 文件的实际位置
        return_location(location);
        // 返回文件的地址
    }

    // ELSE 结束, 非本地

}

// 函数 location-resolution 结束
```

8.2 分布式文件服务

分布式文件服务负责在特定文件上进行的操作。8.2.1 节讲述在分布式环境中文件存在

着多样性，8.2.2 节讨论与文件共享有关的问题，8.2.3 节介绍实现文件服务的方法，在 8.2.4 节中对文件服务主题进行总结并讨论与文件复制有关的问题。

8.2.1 文件多样性

除了在 8.1 节中讨论过的文件类型外，文件可以在存储方式、指派的各种属性、由底层操作系统使用的保护模式等方面发生变化。

文件存储

操作系统可以用各种方式来存储文件。一些系统以连续的字节流来存放文件，这使得在文件中没有内在的结构。这种方法用于大多数现代的操作系统，这种系统支持非结构化的文件。相反，结构化文件（structured files）以如图 8-4 所示的记录结构来表示数据，该文件系统可以或不可以为文件中的单条记录提供索引。如果支持索引，则单个的记录可以显式地通过索引来访问。记录索引通常存放在如 B-树或散列表这样的数据结构上。若不支持索引，则单条记录就只有通过指定记录在文件中的具体位置才能进行访问。

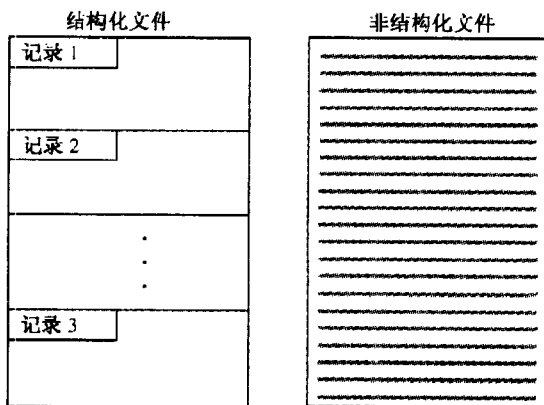


图 8-4 结构化与非结构化文件

此外，文件可以以二进制方式或 ASCII 码方式来存储，一般两种方式都允许使用。二进制文件可以是可执行文件或加密文件（见第 11 章的信息加密），ASCII 码文件包括各种用户可读的数据及源文件。对这两种文件类型的区分，在字长度不能被 8 位整除的系统中非常普遍，如一个 36 位的系统 PDP 10。

文件属性

文件常常有一组属性，提供关于特定文件的额外信息。不同的文件系统为不同的文件属性提供支持，分布式文件系统必须意识到并处理各种类型的属性。一些系统只允许特定的属性子集而另一些系统则允许有灵活性。以下是一组常见的文件属性：

- ◆ 文件名（包括文件类型扩展名）。
- ◆ 文件大小。
- ◆ 文件所有权类型（个人或群体）。
- ◆ 文件所有人名字。
- ◆ 文件创建的日期。
- ◆ 最后一次文件访问的日期。
- ◆ 最后一次修改文件的日期。
- ◆ 版本号。
- ◆ 相关的保护信息（见下文）。

目录服务（在 8.3 节讨论）维护这些属性信息并将它们绑定到特定文件上。

文件保护模式

分布式文件系统必须意识到集中式系统所使用的文件保护模式。集中式文件保护模式是基于文件允许访问的类型、保护所定义的粒度级别,以及文件保护所使用的类型来实现的。一个文件可以通过以下方式访问:

- ◆ 文件读操作。
- ◆ 文件写操作。
- ◆ 文件截断操作。
- ◆ 文件追加操作。
- ◆ 执行文件。

188

保护粒度可以通过单个用户、能具体列出来的用户子集、一组用户、特定组织的所有用户或世界上的所有用户等内容来衡量。

对于集中式系统,有两种主要的文件保护类型:访问列表(access list),也称为访问控制列表(ACL)和权能(capability)。举一个例子来描述这两种保护类型的区别。设想自己是上层社会的一员,被邀请参加一个晚会。主办者可以利用两种方法来保证只有邀请的客人参加晚会。第一种方法是将邀请的客人清单给保安,当人们到达晚会地点时保安将他们的名字与客人清单上的名字核对以检查名字是否在清单上。如果他们的名字在上面,则就允许他们进入。第二种方法是给每位客人发一份请贴,当被邀请者到达时必须将该请贴呈给保安。客人清单类似于文件保护中的访问列表方法,该方法中每个文件维护着一个说明哪个用户被允许以何种方式访问的列表。访问列表由系统来维护。当用户要访问一个文件时,如果用户和操作作为可接受的动作出现在访问列表上,则就许可该操作。同样地,权能同请贴类似,为了访问一个文件用户必须提供一个权能,表明允许他执行请求的操作。

这两种方法同它们的对比物一样有相似的缺点。与晚会的主办者撤消一个请贴时所遇到的困难相同,文件系统不能快速或容易地撤消一个权能。晚会的客人列表可能很大,同样访问列表也非常大。它可以用一个二维数组(列表示文件名,行表示用户名,无论他是否为个人或集体用户,入口栏表示被授予的访问权限)或实际的列表来存储。当访问列表存放在一个二维数组上时,就称为访问矩阵,如图8-5所示。图8-6描述了图8-5中文件1的访问列表。权能和访问列表也可以用来保护和规定对任何系统资源的访问。

	用户 1	用户 2	组 1	组 2
文件 1	RWE	R	R	R
文件 2	R			
文件 3		RWE	RE	RE
文件 4		RWE		RWE
R= 读访问 W= 写访问 E= 执行访问				

图 8-5 访问矩阵

189

访问控制也可以使用权能和访问列表的组合。作为一个例子,让我们参看1996年的奥运会上使用的安全证件。在安全敏感度低的区域,只要出示安全徽章(依据地点和徽章的类型)就可以允许使用需要的资源,例如在公共的通路上骑车。相反,对于安全敏感度高的区域,例如奥林匹克中心或计算机中心,就要仔细检查徽章,并且立即将确切的身份号码同访问列表进行核查。如果没有出示权能——徽章,就不核查访问列表。关于分布式访问控制的额外安全措施将在第11章讨论。

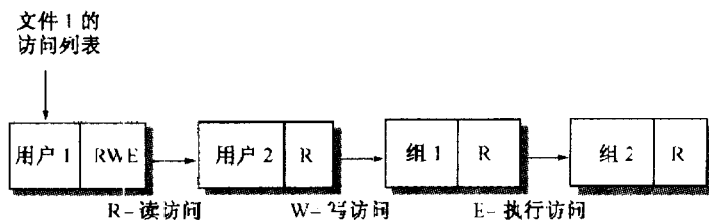


图 8-6 文件 1 的访问列表

8.2.2 文件修改通知

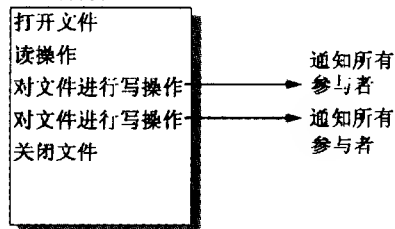
我们已经学过，基本的文件操作包括读、写和执行文件。当多个节点访问共享文件，对于通知其他参与者有关文件修改的方法时每个参与者必须取得一致，这也通常称为文件访问语义（file access semantic）。有两组通知方法，第一组是将所有文件看作不可更改的（immutable）（即不允许修改），由于不允许修改，就不会出现对数据的破坏，这样就不需要通知。第二组通知方法是用于可修改的文件，以下是三种基本的用于可修改文件的修改通知的方法，又如图 8-7 所示：

1. 立即通知：使用立即通知的方法，对文件的每次修改立即对拥有该文件拷贝的所有参与者可见。在分布式系统中这种方法实现起来是非常困难的和不切实际的。

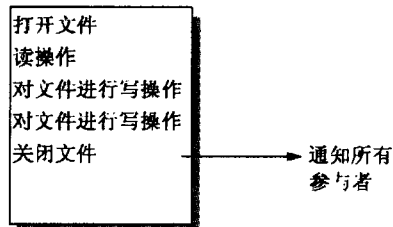
2. 在关闭时通知：使用在关闭时通知的方法，只有对将文件进行修改的参与者关闭文件，也就是终止对文件的访问时，其他参与者才被通知已修改文件，分布式系统中的其他成员没有更新它们的拷贝而是继续使用脏的拷贝。文件的最终内容由最后一个对文件修改的参与者来决定。

3. 在事务完成时通知：一个事务是一组固定的操作，当这组固定的操作完成时，通知系统中的成员。事务的方法在分布式系统中应用得非常成功，有关该主题的讨论见第 9 章。

A. 立即通知



B. 关闭时通知



C. 事务完成时通知

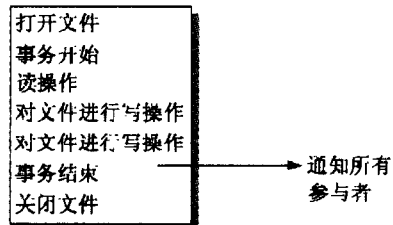


图 8-7 修改通知的方法

8.2.3 文件服务实现

文件服务实现可以是基于远程访问或远程拷贝的，也可以是有状态或无状态的。当使用远程访问时，远程的客户机必须通过网络通信来远程地执行读、写、打开、关闭、追加、截断和处理每个它所希望的操作。客户方不需要文件的任何存储空间，但是必须处理不间断的

网络延迟。客户方不接收文件的实际的拷贝，服务器必须管理特定文件的拷贝，而且可能是大量的。相反，如果文件服务利用远程拷贝来实现，客户机接收被请求文件的一个完整拷贝。这种实现也称为**全文件缓存**（whole file caching）。使用远程拷贝的实现一般只支持读和写操作并要求客户方有足够的存储空间来存放文件。遇到的惟一的网络延迟是在开始和结束时的文件传输。这种方法比较简单，并且当整个文件将被客户方用来读或修改操作时非常成功。一般来说，文件修改通知所使用的方式是文件关闭时通知。应当注意到，如果客户方只需要以块来衡量的文件的一小部分，就会浪费掉网络资源和时间。以文件块为单位来传输的实现称为**块高速缓存**（block caching）。如果远程拷贝允许对文件若干部分进行访问，例如块高速缓存，那么服务器和整个文件系统负责管理的单元数目将大大地增加。

有状态的服务器需要维护访问该服务器文件的所有客户机的信息。具体地说，服务器维护有关客户方使用的是哪个文件，以及对该文件执行过哪些操作的信息。如果某客户机正在读一个文件，有状态的服务器将保存客户机访问的文件的当前位置信息。这将极大地提高从开始与服务器连接传输信息之后的所有信息传输操作的效率，但同时也会增加客户方就某特定文件访问对该服务器的依赖，因此这种系统不能支持故障透明。相反，一个无状态的服务器不维护客户方的任何信息。客户方的任何请求都必须包含特定的请求信息，如文件名、操作、在文件中的确切操作位置。客户方自己维护状态信息，虽然这种方法增加了每个操作的开销，但是它提供了支持故障透明的能力。

8.2.4 文件复制

当分布式文件系统支持文件复制时，文件服务的可靠性和效率将会极大地增强。效率的提高体现在响应时间、服务器负荷，以及网络传输等指标上。此外如果一个服务器崩溃了，文件服务仍然能履行它的义务。只有在所有包含特定文件拷贝的服务器都失效时，该文件的服务才会停止。这样文件复制提供了故障透明性，因为它允许用户继续工作而无需知道某系统是否失效。最初的文件复制方法是显式拷贝，即用户发出一个远程拷贝命令，接着它就接收到文件的拷贝。这种方法为使用如 rcp（远程 UNIX 拷贝）和 ftp（与 TCP/IP 有关的文件传输协议）命令的早期网络所使用。显式的文件复制方法对于一个网络操作系统来说可能已经足够了，但它与分布式操作系统努力提供统一视图的目标有着直接的冲突。分布式操作系统中成功的文

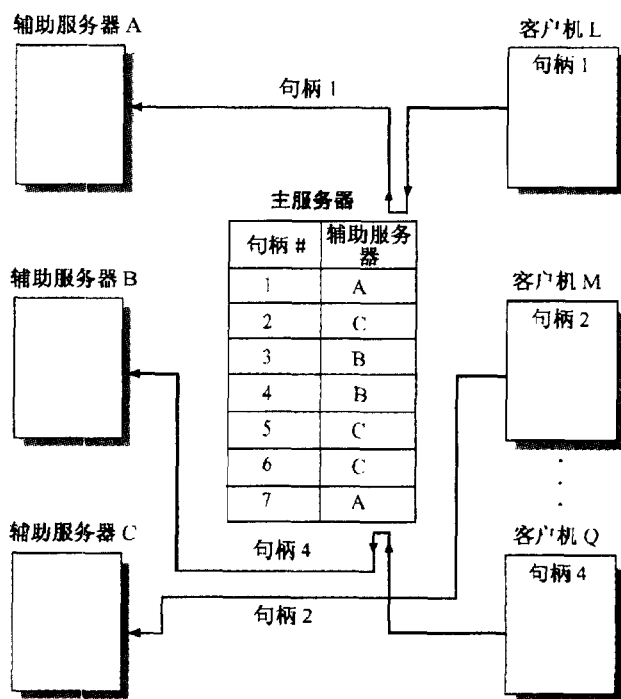
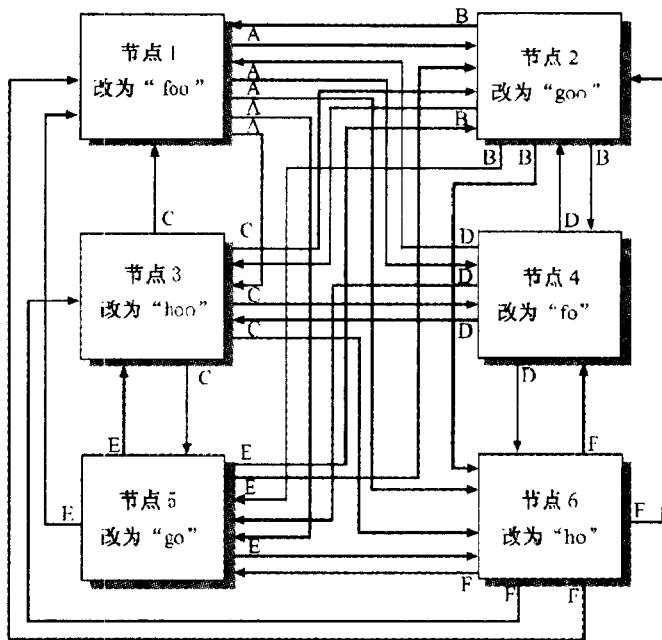


图 8-8 使用中间文件句柄的映射表

件复制必须提供复制透明。文件系统的用户应当意识不到在系统中存在多个文件拷贝，而且任何文件复制方法都必须提供某种形式的数据一致性（见第9章）。

192 分布式系统中的文件复制方案可以是集中式的或分布式的。集中式的方案涉及到指定一个文件服务器为一组文件的主服务器，所有数据更新的请求由这个主服务器来处理。当主服务器失效时，文件更新将不能进行，但通过辅助服务器仍然可对文件进行读操作。主服务器可以给客户方提供辅助服务器上拷贝文件的一个直接的文件句柄（file handle），或提供给所有的客户方一个中间的文件句柄。如果提供中间文件句柄，则主服务器必须维护一个中间文件句柄映射到实际文件句柄的表，如图8-8所示。文件句柄一般包括文件的位置和访问该文件的必要权限。此外为了帮助实现正确的文件更新，文件句柄可以将序列号结合进来，与序列号有关的问题将在第10章中讨论。

对于文件复制有两种流行的分布式解决方案。第一种方案是利用组通信。当一个参与者无论什么时候改变文件的内容时，它将写命令发送给所有的参与者，如图8-9所示。这种通信可以立即进行，使进行写操作的操作者产生一个短暂的中断，或由一个低优先级的后台进程来完成。同样地，接收者可以立即更新它们的文件拷贝，也可以基于“需要时去更新”的原则更新它们的拷贝。在以前的改变命令被处理之前文件可能又一次被改变（本地或远程）时，立即通信和立即更新有助于将这种情况减到最少。



源节点	映 像
1	A- 对文件的修改为 foo
2	B- 对文件的修改为 goo
3	C- 对文件的修改为 hoo
4	D- 对文件的修改为 fo
5	E- 对文件的修改为 go
6	F- 对文件的修改为 ho

图8-9 使用组通信的分布式文件复制

193 第二种分布式的方法涉及到投票和版本号。客户方从各个服务器中获取修改文件的权限。通过对最新版本号达成一致的多数服务器的许可才能取得修改权限，其中规定这些服务器没有发送出更高的版本号。一个放宽限制的方案是只要从多数的服务器

（不需要对版本号取得一致）中接收到简单的许可，并将接收到的最高的版本号作为当前版本号。无论采用哪种方法，写权限通常比读权限需要更多的票数，当接收到文件时版本号就会增加。

194

8.3 分布式目录服务

分布式文件系统中，分布式目录服务负责全面的文件组织，给用户提分布式文件系统

的接口。我们将在 8.3.1 节介绍可能的目录结构，接着在 8.3.2 节讨论目录管理，最后在 8.3.3 节讨论目录服务操作。

8.3.1 目录结构

第一种目录结构是**层次目录结构**。这种目录结构允许有目录和子目录。子目录只有一个父目录，这使得用户可以容易地组织他们的文件，但使多个用户的文件共享变得很困难。相反，**无向图目录结构**允许一个目录可以有多个父目录，这提供了方便的目录共享但使得目录管理复杂化，特别是使得确定和删去孤儿文件更加困难，见 8.3.2 节。集中式系统中一个无向图结构的目录服务例子是 UNIX。UNIX 允许文件的硬链接（在同一文件系统内由系统管理员创建）和软符号链接（跨文件系统和用户创建的），这样就创建了多个父目录。在这种环境下，UNIX 允许结点可以是另一个结点的父结点或子结点的**有向目录结构**，这种结构可能会产生类似于缠绕的线球的结构。

8.3.2 目录管理

目录管理与维护活动目录的正确列表及它们下面的文件有关，而且它必须能够确定和删除未引用文件。在一个层次结构的目录系统中，文件只有一个父目录。当父目录被删除时，系统设定或请求所有的子目录和它们下面的文件也同时被删除。这种文件不再被目录引用，它们的资源释放给系统。相反，如果目录结构是无向图，则文件可能有多个父目录，这就产生了何时认为文件未引用的问题。这是非常重要的，因为它表明了什么时候目录服务允许回收该文件的资源。这个问题可以通过给每个文件一个引用计数器来解决。当链接添加到文件中时该文件的引用计数器就增加，相反当链接从文件中删除（由于文件或目录删除）时引用计数器就减少。任何与引用计数器的值为 0 的文件相关资源将被目录服务回收。在 UNIX 中，硬链接影响引用计数但软符号链接不影响引用计数。这样，如果文件的拥有者决定删除一个有符号链接的文件，则该文件就会被立即删除。与此相反，在有硬链接的文件中删除文件之前，必须删除所有与之相关联的硬链接。

195

目录管理也负责目录结构的存储。目录结构可以存放在一个集中式的位置，也可以在整个系统中分布式地存储。分布式目录服务的一个例子是电话目录服务。如果你希望查找一个电话号码，则必须在正确的目录下查找。没有单个的目录服务可以维护整个系统的信息。因为电话系统不提供位置透明性，所以电话目录服务也不提供位置透明性。与文件复制及分布有关的信息也与目录复制及分布有关。具体地说，分布可以基于某种类型的散列函数或属性，比如创建者。

8.3.3 目录操作

目录服务支持以下所有的目录操作或它的一个子集：

- ◆ 创建目录。
- ◆ 删除目录。
- ◆ 重命名目录。
- ◆ 列出目录内容。
- ◆ 管理目录的访问权限。

- ◆ 改变目录的访问权限。
- ◆ 在整个的目录结构内移动目录。
- ◆ 遍历整个目录结构。

任何未支持的但可能被底层的集中式操作系统允许的目录服务应当有某种类型的默认的映射或等价操作。此外目录服务器还负责以下的文件操作：

- ◆ 创建文件。
- ◆ 删除文件。
- ◆ 重命名文件。
- ◆ 移动文件。
- ◆ 查找文件。
- ◆ 复制文件（在新目录下用新的名字）。

196

同目录操作一样，任何未支持的文件操作应当有某种类型的默认映射或等价操作，因为它们可能被底层的操作系统支持。

8.4 网络文件系统

在本节中，我们介绍 SUN 公司的网络文件系统（NFS）版本 3。8.4.1 节介绍 NFS 的文件服务，8.4.2 节介绍 NFS 的目录服务，8.4.3 节介绍 NFS 的名字服务。

8.4.1 NFS 文件服务

NFS 文件服务是无状态的文件服务，并且是基于如 8.2.3 节所述的块高速缓存。NFS 利用使用外部数据表示（见第 2 章）的 RPC 机制（见第 3 章）为该分布式文件系统的用户取得透明的远程访问。分布式系统的成员可以包括从个人计算机到高性能并行计算机。文件访问控制支持使用 Kerberos 鉴别的 DES 密钥加密，见第 11 章。NFS 鼓励所有的服务器都支持所有的文件属性及文件类型，文件名的实际大小由客户方和服务器共同达成的协定来决定。NFS 版本 3 扩展了版本 2 以支持更大的文件，它能支持 64 位大小的文件。RPC 和所有的 NFS 文件系统到本地文件系统的转换工作由 NFS 中称为虚拟文件系统的系统来处理。这个虚拟文件系统位于本地文件系统的上层，并与本地文件系统交互。

当客户请求访问一个文件时，整个磁盘已安装好，文件的初始块传送给客户。具体地说，客户接收到一个句柄，它包括文件系统类型和文件在磁盘中的地址，用以检索磁盘及磁盘上的文件内容。这个句柄可以持续使用，直到服务器撤消发给该客户的句柄。给予客户句柄称为安装，安装算法的概述见详细说明 8.4。安装可以是显式的，这样访问的允许可以是“按需”或是自动的。自动安装涉及到自动系统程序，它在每次系统启动时或客户访问远程文件时用来自动获取文件块。在任何一种情况下，在调用时为获取磁盘上的文件，自动检索利用句柄发送消息给所有的服务器。可将第一个服务器的响应作为磁盘的来源。这样 NFS 可以用来支持位置、复制、和故障透明性。但是这些特点不是 NFS 内在的特性。值得注意的是，确定信息的最新和正确版本的方法还没有实现或支持。没有使用某种形式的锁，将会出现竞争状态。

197

多个文件拷贝的修改通知被后台进程处理。这样通知不是立即发出的，但修改通知的执行优先于文件关闭。文件中被客户修改的部分标记为脏，只有这些脏块才会被传送回服务器。

详细说明 8.4**NFS 安装**

以下是 NFS 安装所涉及到的步骤和远程文件句柄的使用：

1. NFS 安装由客户来进行实例化。这种实例化必须包括远程目录上的信息，用于安全目的的请求者的标识，以及在客户文件系统中放置请求文件的地址。
2. 安装执行一个 NFS 查找操作来定位维护着所需磁盘拷贝的服务器。
3. 将一个 RPC 发送到相应的服务器，请求所需的磁盘句柄。
4. 远程服务器执行相应的安全检查，看客户对磁盘内容的访问是否许可，若是则返回一个句柄给客户。
5. 客户接收到文件句柄。客户系统将表明文件是远程文件的有关文件句柄信息记录到本地的文件系统表上（用同一张表来存放本地文件的信息）。
6. 接着就用该表来访问文件。如果将要访问的文件标记为远程文件，则用表中的信息初始化一个 RPC 的 NFS 文件请求。如果文件是本地的，那么底层操作系统处理请求。

8.4.2 NFS 目录服务

NFS 假定层次结构的目录系统，但同时也支持创建 UNIX 中常见的软链接。如 8.4.1 节所述，整个目录和子目录也可以传送给客户，这也称为安装。NFS 支持的目录操作列表见详细说明 8.5。

198

详细说明 8.5**NFS 目录操作**

以下是 RFC 1813 中介绍的、NFS 支持的目录操作列表：

- ◆ Lookup: 返回文件及其属性信息。
- ◆ Access: 允许显式验证对文件的访问许可。
- ◆ Read: 读文件，包含有一个用于文件结尾检测的布尔变量。
- ◆ Write: 对文件进行写操作（可以指定是否只对文件中需要写的那部分进行写操作）。
- ◆ Create: 创建文件。
- ◆ Remove: 删除文件。
- ◆ Rename: 重命名文件或目录。
- ◆ Rmdir: 删除目录。
- ◆ Mkdir: 创建目录。
- ◆ Mknod: 创建特殊文件。
- ◆ Readdir: 读目录，包含有安全验证的条款。
- ◆ Readdirplus: 读目录及其属性。
- ◆ Fsinfo: 提供文件系统的信息（最大文件的大小、软链接的可用性等）。
- ◆ Fstat: 提供文件系统的统计信息，包括已用的磁盘空间及剩余的空间。
- ◆ Commit: 强迫文件的脏部分写回到服务器中。

8.4.3 NFS 名字服务

NFS 的名字服务是基于 8.4.1 节中所述的文件或目录的安装。NFS 不支持（不能轻易地使其支持）一个全局的名字空间，客户可以通过显式的安装文件请求共享远程的名字空间。当文件或目录安装后，它就作为本地目录结构的一部分呈现给客户。图 8-10 描述了一个名字为“foo”的文件安装在两个不同的节点上。安装的文件或目录的信息在本地可得到，这样就提供了位置透明性。但是由于这种实现，使得 NFS 系统中每个用户有惟一的全局文件系统视图。并且文件在系统中可以有几个“绝对的”名字，这为使用基于 NFS 的文件系统的分布式应用带来了麻烦，因为一个文件会有几个依赖于位置的文件名。

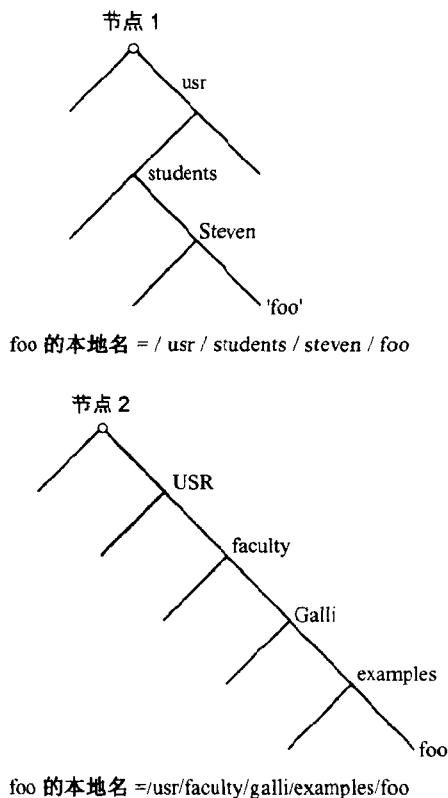


图 8-10 NFS 的名字空间和安装

199

8.5 X.500

在本节中，我们对 X.500 目录服务作一个简要介绍。X.500 也是 Windows 2000 中使用的轻量级目录访问协议（Lightweight Directory Access Protocol, LDAP）的基础，见第 12 章的实例研究。X.500 目录服务是由现在称为 ITU-T（国际电信联盟电信标准化部门）的 CCITT（国际电报电话咨询委员会）和国际标准化组织（ISO）定义的国际标准。该目录服务的设计是用来提供与 Internet 的类似的电话黄页服务及邮件和电子邮件地址查找服务。X.500 的技术报告 RFC 309 中说明的 X.500 目录服务的目标如下：

200

- ◆ 非集中式维护：每个节点只负责维护本地那部分的目录服务，这样存储的信息除本地外均认为是不可修改的。
 - ◆ 支持功能强大的、复杂的查询。
 - ◆ 提供惟一的全局名字空间。
 - ◆ 提供允许本地扩展的结构化的信息框架。
 - ◆ 设计基于标准的目录服务，这样所有的应用能依靠统一的接口来使用目录服务。
- 我们现在介绍 X.500 的文件和名字服务（8.5.1 节）和目录服务（8.5.2 节）。

8.5.1 X.500 文件和名字服务：信息模型

X.500 的文件和名字服务是以信息模型来定义的，信息模型负责信息的存储和安排。设计该规范不是用来实现通用的分布式文件系统的。X.500 信息模型将它的基本结构看作是一个条目，每个条目拥有它所代表的对象的信息。每个对象有一组必备属性和一组可选属性，必备属性总是由对象继承得到。如果一个条目代表一个人，则它可能有以下一组属性：

- ◆ 名。
- ◆ 姓。
- ◆ 邮件地址。
- ◆ 电子邮件地址。
- ◆ 工作地点。
- ◆ 电话号码。
- ◆ 传真号码。

此外每个属性还有相关的语法，用来说明属性所使用的数据类型，如文本或图像。我们以电话号码来说明属性语法，电话号码只许由数字、括号及破折号组成。

X.500 全局名字空间由有着严格层次性结构的目录信息树（DIT）来表示，名字空间中所有的名字及相关的数​​据存放在目录信息库（DIB）中。DIB 的每个条目占据 DIT 中的一个惟一的位置，如果条目没有子条目则它就称为叶条目。每个 DIT 条目和它的属性构成了该条目的相对特异名（relative distinguished name, RDN）。条目完整的特异名由它自己的 RDN 及一直回溯到 DIT 的根条目路径中所有祖先级 RDN 组成。这种方法降低了重复名字的出现概率。例如，一个电话本上可能有几个 S. Smith，如果 S. Smith 的电话本条目包含有回溯 1000 年的母系婚前姓氏，则条目将会是惟一的（假定你能以这种方式来识别出你朋友的名字!）。这样通过将继承包含到特异名中的方式，每个条目将有全局惟一的名字。

201

8.5.2 X.500 的目录服务：目录模型

X.500 的目录服务是以目录模型来定义的，该目录模型假设查询操作比更新操作多很多。目录模型包含的两个组件分别是目录用户代理（DUA）和目录系统代理（DSA）。用户代理代表用户来访问系统，而系统代理维护目录信息库（DIB）的某一部分且能为用户代理提供目录访问点。整个 DIB 由多个 DSA 进行维护。为了支持位置透明和复制透明，DSA 使用两种分别称为链接和参照的技术。DIB 的一个特定部分由一个称为主 DSA 的主节点来维护，存放这同一部分 DIB 的节点由主 DSA 的影子 DSA 来维护。影子 DSA 自己也可以有影子 DSA，但只有主 DSA 才能修改 DIB 的内容。所有报告给影子 DSA 的修改必须传送给该部分 DIB 的主 DSA。

8.6 小结

分布式文件系统有三个主要的任务。第一，它必须提供命名文件的方法及通过名字服务来管理文件名的方法，这可以用支持名字透明的方式来实现。全局命名方案的选择也影响整个文件系统是否支持位置透明。第二，文件系统必须管理文件，包括它们的各种属性及正确的授权访问，这可以通过文件服务来实现。这种服务也负责确定文件是否能被修改。如果

能，文件服务必须确定怎样实现对文件的修改。同时它必须决定当文件被客户使用时文件是仍然在服务器上，还是客户将在本地拥有文件的拷贝。如果允许文件复制，那么提供复制透明也是文件服务的责任。最后，分布式文件系统包括目录服务，目录服务必须提供一个用户接口和文件的整体结构。在分布式系统中必须包括这三种服务的每一种，但这些服务可以作为单独的单元或组合成一个或两个不同的服务来实现。如果目录服务是为了实现分布式操作系统的目标，那么它必须支持所有相关的透明性。

202

在本章中，我们也介绍了两个分布式文件系统的实例。这两个实例有着完全不同的实现方法，每种方法都是突破性的实现。NFS 提供了文件系统的全面的服务，而 X.500 主要是提供目录服务。NFS 提供了全局命名，但同一文件在不同位置名字就不同，而 X.500 提供了一种全局惟一名字的命名方案。NFS 本身不支持复制，而 X.500 利用了全局不可修改数据的特性实现了全面的复制支持。

为了保证可以接受文件内信息修改的行为，分布式文件系统或者利用分布式文件系统的应用也必须解决以下问题。具体地说，它们必须提供并发控制（第 5 章），确定一个合适的一致性模型（第 9 章），以及利用某种形式的序列号（第 10 章）。关于分布式文件系统的研究还在进行当中。当前，轻量级目录访问协议（LDAP）极有可能成为 UNIX 和基于 Windows 系统的统一的解决方案。无论是什么方案，必须解决文件的透明访问问题。随着可用信息数量和需要支持的文件类型（特别是多媒体）的不断增加，提供对世界范围内信息的安全及高效访问是对广大研究人员提出的挑战。

8.7 参考文献

第 1 章中给出的参考文献能提供一些分布式文件系统的总体的信息，涉及到本章内容的一些传统的研究论文包括 [BKT85, CAKLMS92, CBE95, DaBu85, Dio80, FrOl85, GNS88, HKMNSSW88, Kum91, LeSi90, LZCZ86, MiDi82, MSCHRS86, NWO88, San87, Sat90a, Sat90b, ScPu94, SMI80, 和 YSL97]。

以下提供一些 Internet 上涉及到分布式文件系统的资源的链接，对 NFS 作了详细描述 1989 请求评注（request for comments, RFC）可以在 <http://www.cis.ohio-state.edu/htbin/rfc/rfc1094.html> 找到，涉及到 NFS 版本 3 的 1995 RFC 见 <http://www.cis.ohio-state.edu/htbin/rfc/rfc1813.html>，与 NFS 有关的所有 RFC 的完整的清单见 <http://www.cis.ohio-state.edu/cgi-bin/wais-rfc.pl?NFS>。NFS 的另外的信息可以在 SUN 公司的主页 <http://www.sun.com> 上查找到。以下是有关 CCITT 的 X.500 目录服务的一些 RFC：

- ◆ <http://www.cis.ohio-state.edu/htbin/rfc/rfc1308.html>（执行的介绍）。
- ◆ <http://www.cis.ohio-state.edu/htbin/rfc/rfc1309.html>（技术概述）。
- ◆ <http://www.cis.ohio-state.edu/htbin/rfc/rfc1275.html>（复制的要求）。

203

与 X.500 有关的 RFC 的完整清单可以在以下链接获得：

- ◆ <http://www.cis.ohio-state.edu/cgi-bin/wais-rfc.pl?X.500>（与 X.500 有关的所有 RFC 的完整清单）。

AFS 是另一种使用远程拷贝的流行的分布式文件系统，AFS 代表 Andrew 文件系统，Andrew 是该系统的发源地——内基——梅隆大学的创办者。AFS 已经商业化，它现在是 Transarc 公司的一个产品。有关 AFS 的概述见 http://www.alw.nih.gov/Docs/AFS/AFS_toc.html。AFS 在

Transarc 公司中的主页在 <http://www.transarc.com/afs/transarc.com/public/www/Product/AFS>。与 AFS 有关的常见问题 (FAQ) 及回答见 <http://www.transarc.com/afs/transarc.com/public/www/Product/AFS/FAQ/faq>。

LDAP 的 RFC 可以在以下链接中找到:

- ◆ <http://www.cis.ohio-state.edu/htbin/rfc/rfc2251.html>。
- ◆ <http://www.cis.ohio-state.edu/htbin/rfc/rfc2252.html>。
- ◆ <http://www.cis.ohio-state.edu/htbin/rfc/rfc2254.html>。
- ◆ <http://www.cis.ohio-state.edu/htbin/rfc/rfc2255.html>。
- ◆ <http://www.cis.ohio-state.edu/htbin/rfc/rfc2307.html>。

LDAP 的用户手册的在线版本见 <http://www.umich.edu/~dirsvcs/ldap/doc/man/>。此外, 有多个与 LDAP 文档进行链接的且包含有 FAQ 链接的 Web 页地址为 <http://www.umich.edu/~dirsvcs/ldap/doc/>。

习题

- 8.1 分布式名字服务的主要功能是什么? 对于集中式操作系统来说名字服务有必要吗? 如果没必要, 请解释为什么。如果有必要, 阐述分布式名字服务所需要的额外特性。
- 8.2 如果支持文件类型, 在分布式操作系统中当节点使用不同的底层操作系统时会带来什么问题? 请阐述两种支持/处理本地未知类型文件的方案。
- 8.3 请阐述与支持位置透明有关的两个优点和两个缺点 (或问题)。
- 8.4 早期的操作系统有时提供一个平面的名字空间和平面的目录结构, 树形结构是不允许的, 所有的文件都在同一层。在多用户环境下平面名字空间和目录结构的问题是什么? 这是分布式环境下实用的方法吗?
- 8.5 如果要求每个节点都维护一个由名字映射到位置的完整列表, 这样的分布式系统有什么缺点?
- 8.6 请阐述在分布式文件系统中使用复制的三个优点。
- 8.7 哪种类型的应用会从使用结构化文件中受益? 请阐述支持结构化文件的缺点。
- 8.8 根据以下撤消访问权限的可能类型, 评价权能和访问权限。也就是说, 该方法支持相应的撤消吗? 如果它支持, 请说明为什么支持。如果它不支持, 请阐述一个使它支持的解决方案。陈述任何你所作的假设。
 - A. 立即撤消。
 - B. 只对单个客户撤消权限。
 - C. 对单个客户部分地撤消权限 (例如撤消写权限但保留读权限)。
 - D. 暂时撤消权限。
- 8.9 请阐述只支持不可修改文件的分布式文件系统的优缺点。
- 8.10 依据以下项目, 比较有状态的和无状态的文件服务器:
 - A. 支持故障透明。
 - B. 支持位置透明。
 - C. 支持复制透明。

D. 效率。

8.11 依据以下项目，比较基于远程访问的文件服务器和远程拷贝的文件服务器：

A. 支持故障透明。

B. 支持位置透明。

C. 支持复制透明。

D. 修改通知。

E. 文件修改管理。

F. 对服务器的要求。

G. 对客户的要求。

H. 效率。

8.12 在分布式文件系统中，使用中间文件句柄与使用直接文件句柄相比其优点是什么？

8.13 请阐述在分布式系统中支持无向图目录结构所面临的困难，这种结构对全局命名方案有什么影响？又对允许软链接有何影响？

8.14 NFS 支持位置透明吗？为什么？请详细说明你的理由。

8.15 为什么 NFS 支持全局名字空间很困难？请讲述一种让 NFS 能支持全局名字空间的可行方案。

8.16 请阐述 X.500 假定查询操作比更新操作出现的机会多得多的目的。X.500 能够避免什么问题而 NFS 却不能避免？

8.17 在分布式文件系统中提供同 X.500 一样的非集中式文件系统维护的优点是什么？

8.18 允许 X.500 的影子 DSA 自己有影子 DSA 的优点是什么？

8.19 为什么使用分布式文件系统的应用必须确保在某一层次上使用并发控制机制？

8.20 就分布式文件系统所支持的三种服务来比较 NFS、X.500 及 AFS（在 8.7 节有它的相关链接）。

205

206

第9章 事务管理和一致性模型

在前几章讨论的一些内容中，我们经常需要允许分布式系统中的多个成员访问和改变共享的数据。在这一章中，我们将集中探讨一些有关确保正常的数据库操作的问题。第5章里我们学习了几种并发控制的方法。在这一章中，我们要集中探讨在分布式系统中操作共享数据最成功的方法之一：事务管理。事务管理可以同 DSM 或者分布式文件系统合并，用一些我们学过的并发控制的机制去实现。在9.1节中我们将看到一些合并的事务管理在分布式系统中的例子。在9.2节中我们将给出事务的定义及其基本的 ACID 特性。9.3节将介绍一致性的多种形式，它们可能是由事务管理系统和 DSM 提出的。9.4节将讨论一个常见的实现事务管理的协议。9.5节将介绍嵌套事务的概念。最后，9.6节将讨论和实现事务管理系统相关的许多问题。

207

9.1 事务管理的动机

事务 (Transaction) 就是对于共享数据的一组操作的集合，这组操作可以作为单个的操作。**事务管理**就是为确保正常的事务操作而提供的服务。有人可能会想起**竞争条件**，即一个函数的输出或者一个变量的值取决于输入的时间顺序。特别地，两个输入可能为了确保它们的结果是最新的而造成竞争。事务就是来处理竞争条件在分布式环境下产生的问题的。

有很多事情都会驱动人们，并引起他们的兴趣。一个普通的方法是谈论钱，可能“金钱万能 (Money talks)”一词就是由此而来的。这一节就要展示如果缺乏正确的事务管理将产生的两个潜在问题：更新遗失和恢复不一致。我们利用银行账户作为一个例子，如果没有事务管理，这个例子中的账户就是你的！

9.1.1 更新遗失

在这一节中，我们介绍一个促进事务管理的经典问题——更新遗失。**更新遗失**这一词汇用在本书内是指不正确地处理了数据库操作，以致某一动作的结果遗失了。为了说明更新遗失问题，我们考虑以下两个动作。

动作 1

Lynn 有一张“现金卡”可以自动存取她在银行的共享账户。她在自动取款机上要求取出\$200。

动作 2

Steven 也有一张“现金卡”可以自动存取 Lynn 的银行账户，此账户是他和 Lynn 共享的。他在自动取款机上要求取出\$300。

现在假设 Steven 和 Lynn 同时在不同的自动取款机上操作。现在账户余额是\$2500。如果自动取款机上安装的程序在分布式环境中不能正确运行，以下场景之一就可能发生。这个例子的伪码表述见详细说明 9.1。

场景 1

两个自动取款机都得到账户余额为\$2500的信息。Lynn的动作先完成。结果，\$2300的余额写回账户。过了一会儿，Steven完成了他的动作，然后\$2200的余额写回账户，如图9-1所示。而正确的余额应为\$2000。

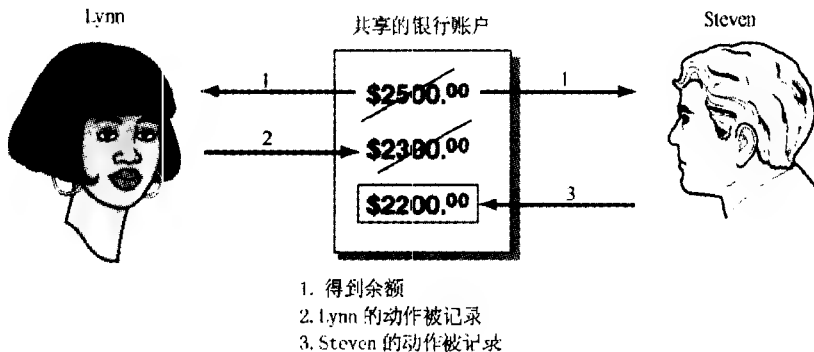


图 9-1 更新遗失

场景 2

两个自动取款机都得到账户余额为\$2500。Steven的动作先完成。结果，\$2200的余额写回账户。过了一会儿，Lynn完成了她的动作，然后\$2300的余额写回账户。而正确的余额应为\$2000。

你可能正在想：“嘿，太棒了！Lynn和Steven赚钱了！”但是如果他们是去存款呢，又或者你是站在银行一方的立场呢？没有一个场景是正确或者说是可以接受的。在分布式环境中的数据操作应该不仅允许动作一起发生，还要提供方法来保证正确的账户余额，本例中应为\$2000。

9.1.2 检索的不一致

当一个数据值返回时，我们都认为它是正确的。尽管这肯定是一个令人期望的属性，但是如果没有事务管理对数据的所有拷贝进行正确处理，它就不一定成立了。以下第一个条件或者两个条件都满足，将可能产生恢复不一致的问题。

- ◆ 共享数据的修改操作不是原子性的。
- ◆ 存在数据的多个拷贝。

详细说明 9.1

更新遗失

动作 1
Get Balance;
Balance=Balance-200;
Return Balance;

动作 2
Get Balance;
Balance=Balance-300;
Return Balance;

场景 1

```

ACTION1: Get Balance;           // 账户余额为$2500
ACTION2: Get Balance;           // 账户余额为$2500
ACTION1: Balance=Balance-200;
ACTION2: Balance=Balance-300;
ACTION1: Return Balance;
ACTION2: Return Balance;
RESULT: Balance=2200

```

场景 2 (图 9-1)

```

ACTION1: Get Balance;           // 账户余额为$2500
ACTION2: GET Balance;           // 账户余额为$2500
ACTION1: Balance=Balance-200;
ACTION2: Balance=Balance-300;
ACTION2: Return Balance;
ACTION1: Return Balance;
RESULT: Balance=2300

```

恢复不一致包含了检索到的数据不能反映最新的事务操作情况。当共享数据的修改操作非原子性时，不但更新遗失问题会发生，而且恢复不一致的问题也会发生。如果数据有多个拷贝，而且这些拷贝也不是原子性地更新时，这问题就更容易发生了。更新遗失和恢复不一致之间主要的一点不同可以用一个词来概括：时间。在更新遗失中，两个动作是在同时发生的。恢复不一致则可以在事务开始后的任何时间发生，只要数据的拷贝没有在另一个动作取得数据值之前更新。我们现在来看看恢复不一致问题上时间因素的一个例子。我们再一次应用 9.1.1 节中的动作 1 和动作 2。

210

动作 1

Lynn 有一张现金卡可以自动存取她在银行的共享账户。她在自动取款机上要求取出 \$200。

动作 2

Steven 也有一张现金卡可以自动存取他与 Lynn 共享的银行账户。他在自动取款机上要求取出 \$300。

现在假设 Lynn 和 Steven 在不同的自动取款机上操作，尽管其中一个比另一个稍稍晚了一点，但两人几乎是同时进行操作的。当前账户的余额为 \$2500。假定自动取款机上没有设置成能在分布式环境中正确操作，那么以下任一场景就有可能发生。这些例子的伪码方式在详细说明 9.2 中给出。

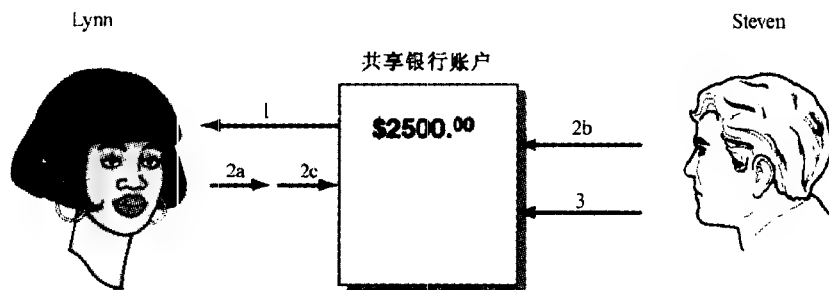
场景 1

Lynn 首先来到自动取款机处。她检索到的余额为 \$2500。从这个余额中，她取出了 \$200，剩下的余额为 \$2300。在正确的余额传送到系统之前，Steven 来到了自动取款机处，他检索到了一个不正确的账户余额 \$2500，这个余额没有反映 Lynn 的操作效果。Steven 从这个余额中提出了 \$300，记录下的余额为 \$2200。注意，虽然记录了 Lynn 的动作，但是恢复不一致导致了一个对于共享数据的不正确的操作，如图 9-2 所示。

211

场景 2

Steven 首先来到自动取款机处。他检索到的余额为 \$2500。从这个余额中，他取出了 \$300，



1. Lynn 检索余额并取出 \$200
- 2a. Lynn 事务的结果开始向银行账户记录发送
- 2b. Steven 检索余额并取出 \$300 [恢复不一致发生了!]
- 2c. Lynn 的结果传输到银行记录
3. Steven 的结果传输到银行记录

图 9-2 恢复不一致

详细说明 9.2

恢复不一致

动作 1

```
Get Balance;
Balance=Balance-200;
Return Balance;
```

动作 2

```
Get Balance;
Balance=Balance-300;
Return Balance;
```

场景 1

```
ACTION1: Get Balance;
```

// 账户余额为 \$2500

```
ACTION1: Balance=Balance-200;
```

```
ACTION1: Return Balance;
```

```
ACTION2: Get Balance;
```

// 账户余额为 \$2500 —— 不正确的值

```
ACTION2: Balance=Balance-300;
```

```
ACTION2: Return Balance;
```

```
RESULT: Balance=2200
```

场景 2

```
ACTION2: GET Balance;
```

// 账户余额为 \$2500

```
ACTION2: Balance=Balance-300;
```

```
ACTION2: Return Balance;
```

```
ACTION1: Get Balance;
```

// 账户余额为 \$2500 —— 不正确的值

```
ACTION1: Balance=Balance-200;
```

```
ACTION1: Return Balance;
```

```
RESULT: Balance=2300
```

剩下的余额为\$ 2200。在正确的余额传送到系统之前, Lynn 来到了自动取款机处。她检索到了一个不正确的账户余额\$ 2500, 从这个余额中提出了\$ 200, 记录下的余额为\$ 2300。注意, 虽然记录了 Steven 的动作, 但是恢复不一致又一次导致了一个对于共享数据的不正确的操作。

9.2 事务的 ACID 特性

更新遗失和恢复不一致的问题只是在分布式系统中操作共享数据时可能产生的问题中的两个例子而已。另一个问题是在数据操作过程中服务器的崩溃。在所有的场景中, 把一组动作定义为一个事务以及应用事务管理将有助于确保正确的数据操作, 即便系统崩溃也是如此。回想一下, 事务是一组相关联的动作的集合, 事务管理指的是在分布式系统中对所有事务的恰当操作。Harder 和 Reuter 在 [HaRe83] 中建议用助记符 ACID 来记忆事务的基本特性, 如下所示:

- ◆ 原子性 (Atomicity): 事务必须保证它要么完整地发生要么完全不发生, 即便存在故障, 事务也应该是完整的或者索性没有。
- ◆ 一致性 (Consistency): 事务必须以一致的状态开始, 并在一致的状态中离开系统。9.3 节将详细讨论一致性的多种形式。
- ◆ 独立性 (Isolation): 除了要求事务必须以完整的或没有的方式发生以外, 还应要求其其他的成员一定不能访问中间的结果。所有中间的操作都独立地执行, 外界成员的访问只有在满足一致性条件时才能进行。
- ◆ 持久性 (Durability): 一旦服务器提交了一项事务, 它就需要完成这项事务, 即便遇到了系统故障, 同时, 这些结果也应成为永久结果。

现在我们来检查一下为什么合适的事务不会受更新遗失的影响。首先, 事务必须是原子性的。这样, 分布式系统中的多个成员就不会在想改变账户余额的时候并行地检索到账户的相同余额。其次, 根据一致性特性, 上述场景都不会发生, 因为事务离开系统的时候均处于不正确的、而且是不一致的状态下。第三, 根据独立性特性, 一个动作不能干涉别的动作。在所有的实际情况中, 一个动作必须等待其他的动作完成。最后, 根据持久性特性, 动作必须是持续的。相反, 如果在更新遗失的例子中 Lynn 的动作先完成, 则她的动作不持久, 也不一致。

我们现在检查一下这些 ACID 特性能否防止上述 Steven 和 Lynn 遇到的恢复不一致问题。首先, 如果动作是原子性的, 则读出余额、改变余额和写回余额将作为一个单独的动作出现。所以, 第二个动作就不能在这个单独动作中间发生。这一点也同时由独立性要求所保证; 然而最主要的特性还是一致性特性。在不能保证动作原子性的条件下, 事务在一个非一致的状态中离开了系统。特别地, 如果系统是在一个一致的状态下, 而且也符合持久性特性, 恢复不一致问题就不会发生。所以, 如果应用了正确定义的事务概念, 只会发生以下的两个场景。在详细说明 9.3 中给出了事务的伪码表示。

事务 1

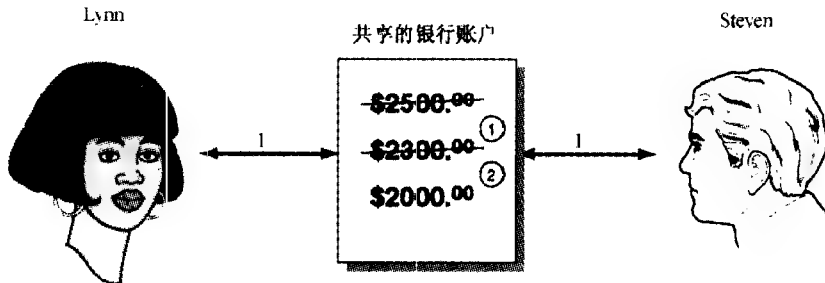
Lynn 有一张能自动存取她的共享银行账户的现金卡, 她要求取出\$ 200。

事务 2

Steven 也有一张现金卡可以自动存取他的银行账户, 此账户是和 Lynn 共享的。他要求取出\$ 300。

场景 1

Lynn 的事务首先执行，执行的结果余额 \$ 2300 被永久记录下来。Steven 的事务随后执行，一个新的执行结果余额 \$ 2000 也被永久记录下来，如图 9-3 所示。



1. Lynn 的原子事务发生。检索得到 \$2500 的余额，取出 \$200，然后永久记录新的账户余额 \$2300。
2. Steven 的原子事务发生。检索得到 \$2300 的余额，取出 \$300，然后永久记录新的账户余额 \$2000。

图 9-3 正确的事务管理

场景 2

Steven 的事务首先执行，执行的结果余额 \$ 2200 被永久记录下来。现在执行 Lynn 的事务，一个新的执行结果余额 \$ 2000 被永久记录下来。

214 无论是哪一个场景，应用遵循 ACID 特性要求的事务保证了正确的结果。

详细说明 9.3

事务

事务 1

```
Get Balance;
Balance=Balance-200;
Return Balance;
```

事务 2

```
Get Balance;
Balance=Balance-300;
Return Balance;
```

场景 1

```
Begin TRANSACTION1
TRANSACTION1: Get Balance;
```

// 账户余额为 \$2500

```
TRANSACTION1: Balance=Balance-200;
TRANSACTION1: Return Balance;
```

// 账户余额为 \$2300

```
End TRANSACTION1
```

```
Begin TRANSACTION2
```

```
TRANSACTION2: Get Balance;
```

// 账户余额为 \$2300

```
TRANSACTION2: Balance=Balance-300;
```

```
TRANSACTION2: Return Balance;
```

```
End TRANSACTION2
```

```
RESULT: Balance=2000
```

场景2

```
Begin TRANSACTION2
TRANSACTION2: GET Balance;           //账户余额为 $2500
TRANSACTION2: Balance=Balance-300;
TRANSACTION2: Return Balance;        //账户余额为 $2200
End TRANSACTION2
Begin TRANSACTION1
TRANSACTION1: Get Balance;           //账户余额为 $2200
TRANSACTION1: Balance=Balance-200;
TRANSACTION1: Return Balance;
End TRANSACTION1
RESULT: Balance=2000
```

215

9.3 一致性模型

我们已经学过，一致性是事务要求的一个特性。在第4章中我们讨论共享数据操作的多种形式时曾经遇到过这个术语。这一节中，我们要检查一致性模型的多重形式。每种一致性模型都有其复杂度，复杂度较高的模型可能会对系统的性能产生负面影响。一旦选择了一个一致性模型，在分布式系统中的所有成员，包括程序、DSM以及目录服务都必须遵循这个模型。我们从最严格、最复杂的模型开始介绍，到仅有较少限制的模型结束。

9.3.1 严格一致性模型

严格一致性模型 (strict consistency model) 对事务管理系统所需要确保的内存一致性有着最强的要求。有一点很明确，读操作所返回的值必须总是反映最近更新的结果。也就是说，它必须返回最近一次写操作所写入的值，而无论是谁执行了这一项写操作，也无论是在分布式系统中的何处执行。为了在大量数据中识别最近的值，就需要有一个全局时间概念。所以，要实现严格一致性模型，就必须采用在第10章中提到的某一个全局时间方法。

即便有了创建全局时间的方法，要求在分布式系统中采用严格一致性也不明智。具体地说，即使系统相信它找到了最近更新的值，在这个值返回客户端之前的瞬间，此数据的一个新值仍然可能写入。可能当这个刚刚“过期”的值开始向网络上的目的地传输的时候，这个值已经重写入。所以，返回目的地的值已经不是最近的值了，严格一致性也被破坏了。这样，一个在集中式环境中理所应当的一致性模型在分布式环境中实际上并不能够实现。

9.3.2 顺序一致性模型

顺序一致性模型 (sequential consistency model) 对一致性的要求稍微弱一点，但更具实际意义。它首先由 Lamport 提出 [Lam79]。顺序一致性模型要求分布式系统中所有成员和它们的进程共享一个通用视图，此视图记录了对于共享内存访问操作的顺序。它不保证得到的值是最近更新的结果，所以，相同程序的两次运行有可能得到不同的输出结果。

为了理解顺序一致性，我们再回顾一下9.2节中的例子。两个场景都是符合顺序一致性的场景，但是，一些成员看到场景1而同时另一些成员却看到场景2是不允许的。如果一

216

个成员看到 Lynn 的事务首先执行，那么在顺序一致性模型中所有的成员都将共享这个视图。

顺序一致性和严格一致性之间主要的不同是对于全局时间的依赖。顺序一致性并不依赖一个全局时间，而更强调各事件保持同一的顺序。这个方法虽然在编写应用程序时非常直观，但是受到一个重要的性能问题的困扰——读写性能之间是反关联的。任何提升读操作性能的同时也降低了写操作的性能，反之，提升写操作性能的改进也会降低读操作的性能 [LiSa88]。当然，如果所用的应用程序绝大部分操作只是读操作的话，这样的关系可能就是有用的特征了。

9.3.3 偶然一致性模型

偶然一致性模型 (casual consistency model) 是由 Hutto 和 Ahamad 首先提出的 [HuAh90]。这个模型比顺序一致性更进一步放宽了对一致性限制。也就是说，不是所有事件都必须以相同的次序出现在分布式系统中的所有成员面前，只有那些偶然相关 (casually related) 的事件才需要。如果事件是偶然相关的，那么它们就必须以相同而且是正确的次序出现。这使得我们有必要给偶然相关一个定义，一个偶然相关的事件影响会与其偶然相关的事件。如果事件将修改同一个共享数据集，那么它们就是偶然相关的。

举例来说，如果你读出余额并写入一个新的余额，那么这里的读写操作就是偶然相关的。这样，此处的读写操作不但要以相同的次序出现在所有人面前，而且读操作一定要出现在写操作的前面。利用 9.2 节的例子，Lynn 的读写操作相互间是偶然相关的，Steven 的读写操作相互间也是偶然相关的。Lynn 和 Steven 的事务是偶然相关的，因为谁的事务后发生反映了两个事务发生的结果。如果 Susan 有一个在相同银行的不同账户，在她的账户上操作她的事务，她的事务和 Lynn、Steven 的就完全没有关系，如图 9-4 所示。Susan 的事务和 Lynn、

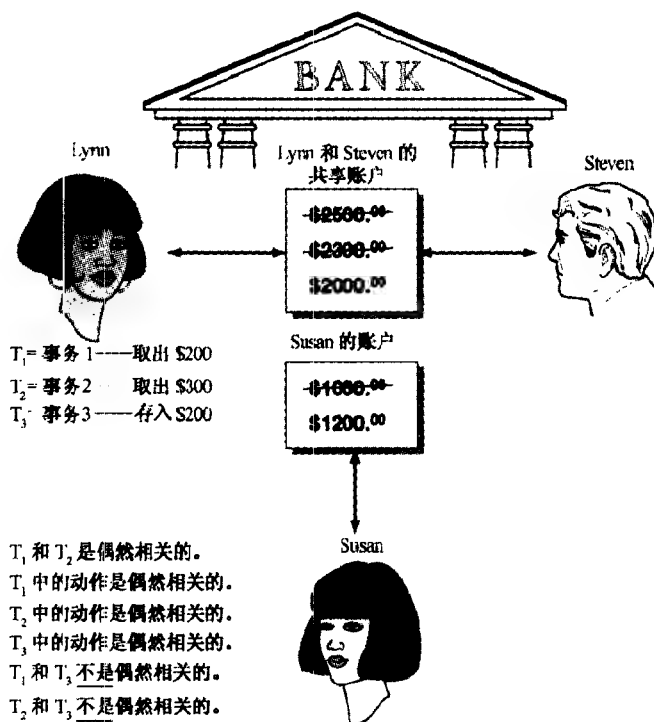


图 9-4 偶然一致性例子

Steven 的就不是偶然相关的。因此,偶然一致性只是在相关的事件间增加了次序的约束。这种一致性模型经常采用几种不同的关系图来表示。这种有时称为依赖图的关系图必须记录哪一个进程看到了哪一个写操作,以此来记录偶然关系。在关系图中没有偶然关系的事件就是无关的事件,也就不受这种一致性模型的约束。这些无关的事件可以看作是并行的操作,并行操作可以认为是在不同的地方以不同顺序发生的。

217

9.3.4 PRAM 一致性模型

管道随机存取内存 (pipelined random access memory, PRAM) 一致性模型最早是由 Lipton 和 Sandberg 首先提出的 [LiSa88]。这个模型比偶然一致性对一致性的要求更弱一些。明确地说,只有一个进程内的写操作才被要求以它们执行的顺序出现在其他进程均能看到的视图中,如图 9-5 所示。每个进程可以想象这些操作处于一个先进先出的队列或管程中,注意这里没有提到关于其他成员的顺序。每一个成员可以以任意次序看到这些来源于多个位置的操作,但是从同一位置来的操作之间的次序必须保证不变。也就是说,在每个地点看来,从同一位置来的操作之间的事务顺序是相同的,而它们和系统中其他来自不同位置的操作之间的顺序则因观察地点的变化而有可能不同。结论就是不同位置在视图图中是允许并行操作的。

218

不正确的 PRAM 一致性

进程 1		进程 2	
本地动作	全局视图	本地动作	全局视图
读 foo	(P_1, W_1)	读 goo	$(P_1, W_2)^*$
写 $foo(P_1, W_1)$	(P_1, W_2)	写 $goo(P_2, W_1)$	$(P_1, W_1)^*$
写 $foo(P_1, W_2)$	$(P_1, W_2)^*$	写 $goo(P_2, W_2)$	(P_2, W_1)
	$(P_2, W_1)^*$		(P_2, W_2)

写操作必须以相同的顺序在所有的进程前出现

正确的 PRAM 一致性

进程 1		进程 2	
本地动作	全局视图	本地动作	全局视图
读 foo	(P_1, W_1)	读 goo	(P_2, W_1)
写 $foo(P_1, W_1)$	(P_2, W_1)	写 $goo(P_2, W_1)$	(P_2, W_1)
写 $foo(P_1, W_2)$	(P_1, W_2)	写 $goo(P_2, W_2)$	(P_1, W_1)
	(P_2, W_2)		(P_2, W_2)

写操作以相同的顺序在所有的进程前出现

图 9-5 PRAM 一致性模型举例

9.3.5 处理器一致性模型

Goodman 提出的处理器一致性模型 [Goo89] 与 PRAM 一致性非常类似,有时还容易混淆,实际上它的限制更紧些。它的要求和 PRAM 一致性一样,还要加上一个存储一致性的条

件。所以，不仅给定位置的所有操作必须管程化，而且对于任意一个特定的存储位置，整个系统必须就此位置的所有写操作顺序达成一致，如图 9-6 所示。Goodman 没有给出在处理器一致性模型中对读访问顺序的要求，Ghorechorloo 等人给出了以下两个条件 [GLLGGH90]。

219

不正确的例子

进程 1		进程 2	
本地动作	全局视图	本地动作	全局视图
读 foo	(P_1, W_1)	读 foo	(P_2, W_1)
写 foo(P_1, W_1)	(P_2, W_1)	写 foo(P_2, W_1)	(P_1, W_1)
写 foo(P_1, W_2)	(P_1, W_2)	写 foo(P_2, W_2)	(P_2, W_2)
	(P_2, W_2)		(P_1, W_2)

因为所有的写操作是对于同一位置的，
所以必须以相同顺序在视图中出现！

正确的例子

进程 1		进程 2	
本地动作	全局视图	本地动作	全局视图
读 foo	(P_2, W_1)	读 foo	(P_2, W_1)
写 foo(P_1, W_1)	(P_2, W_2)	写 foo(P_2, W_1)	(P_2, W_2)
写 foo(P_1, W_2)	(P_1, W_1)	写 foo(P_2, W_2)	(P_1, W_1)
	(P_1, W_2)		(P_1, W_2)

访问同一内存位置的相同全局视图。

图 9-6 处理器一致性例子

1. 一个在其他位置执行的读操作必须等它之前的读访问都完成后才能执行。
2. 一个写操作必须等它之前的读写操作都完成后才能执行。

这些条件允许一个在写操作之后的读操作绕过写操作执行。Ghorechorloo 等人指出，为了防止死锁，使用处理器一致性模型的实现必须保证所有的写操作最后都能被执行。

9.3.6 弱一致性模型

弱一致性模型是由 Dubois 等人提出的 [DSB86, DSB88]。这个模型的定义要用到一种新的变量，即用作同步存储的同步变量 (synchronization variable)。同步变量用作向其他机器传播写操作，以及对于全局数据在分布式系统中其他地方出现的修改作本地更新，如图 9-7 所示。因此，同步变量可以作为组存储的引用。利用同步变量，我们现在来定义弱一致性的三个要求。

220

1. 访问同步变量要遵循顺序一致性模型。
2. 对同步变量进行的访问必须等它之前的写操作在分布式系统中的所有地方都执行了

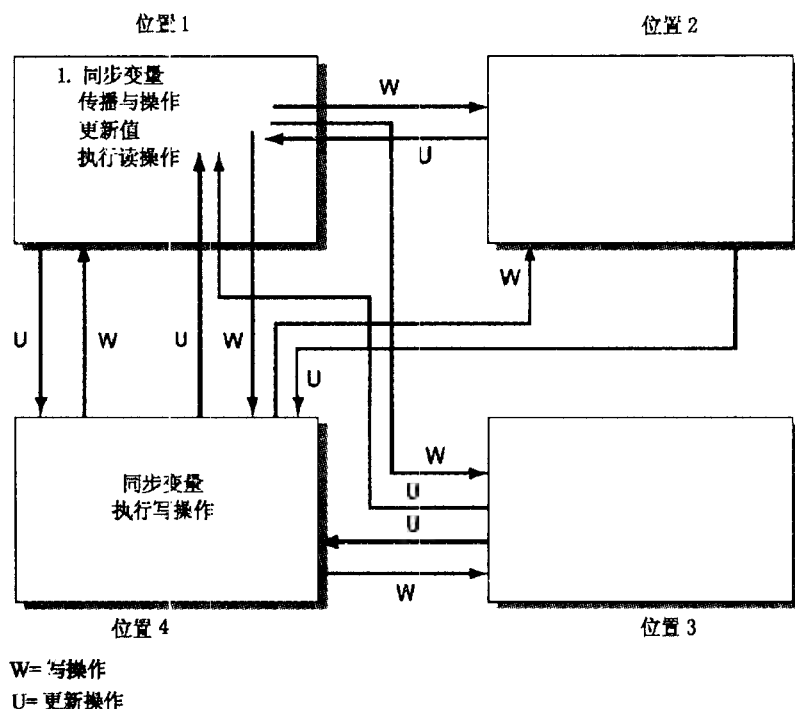


图 9-7 在弱一致性中应用同步变量

后才能进行。

3. 对非同步变量的访问（甚至只是读操作）只有在之前的所有对同步变量的访问都完成后才能进行。

因此，第1点说明所有分布式系统中的进程共享一个记录同步变量访问操作顺序的通用视图。回想一下，顺序一致性模型并不保证所获得的值是最近更新的。然而第2点要求所有位置完成所有正在处理或者已处理完但还没有写的写操作。这样，通过在读操作前执行同步，程序就能确保它从内存中取出的值是最新的。如果在修改共享数据后执行同步，程序就能将新的值传递到系统中各个位置。由于第3点防止了在使用同步变量时对非同步变量的访问，也可以通过在访问共享数据前执行同步来确保最新的值。

这个模型比其他模型具有更好的性能，具有以下优点：

- ◆ 分布式系统中许多（经常是绝大部分）成员不需要看到所有的写操作。
- ◆ 分布式系统中许多（经常是绝大部分）成员不需要看到中间的写操作。实际上，如果这些中间结果是由于在临界区中的操作引起的话，就根本不应看到。

最后，弱一致性利用顺序一致的同步变量对内存引用进行分组。由于同步变量所引起的额外开销，相对于寻常的、单独的逐个存储操作，聚集地访问共享数据，并且每次聚集都执行许多存储操作能取得更高的性能。如果程序中有很多单独的独立的存储操作，它就不能很好地利用弱一致性所给予的成组存储引用的优点。尽管弱一致性能提高性能，但同时它也增加了对使用这一模型的开发者、设计者和程序员的负担，因为操作局部或者全局同步变量将牵涉到所有共享变量的更新。系统不能分辨访问同步变量进行同步操作到底是为了在系统中读数据还是为了分送一个新写入的值。两种情况下，弱一致性都要分送所有的写操作，并更

新数据的本地拷贝。

9.3.7 释放一致性模型

释放一致性模型是由 Gharachorloo 等人提出的 [GLLGGH90]，与弱一致性有些类似，但是释放一致性具有了分辨进入临界区前的同步动作和退出临界区后的同步动作的能力。也就是说，释放一致性提供了**获得访问权**（acquire access）和**释放访问权**（release access）。获得访问权用来通知系统进程正准备进入临界区，这种情况下，所有其他成员所作的修改结果都要加以传播，并由本地的处理器来更新。释放访问权是用来通知系统，进程正在退出临界区，对共享内存所作的本地修改将传播到其他的各个成员。这使得释放一致性只要对每一种特定的同步操作采取适当动作就能获得比弱一致性更高的性能，如图 9-8 所示。

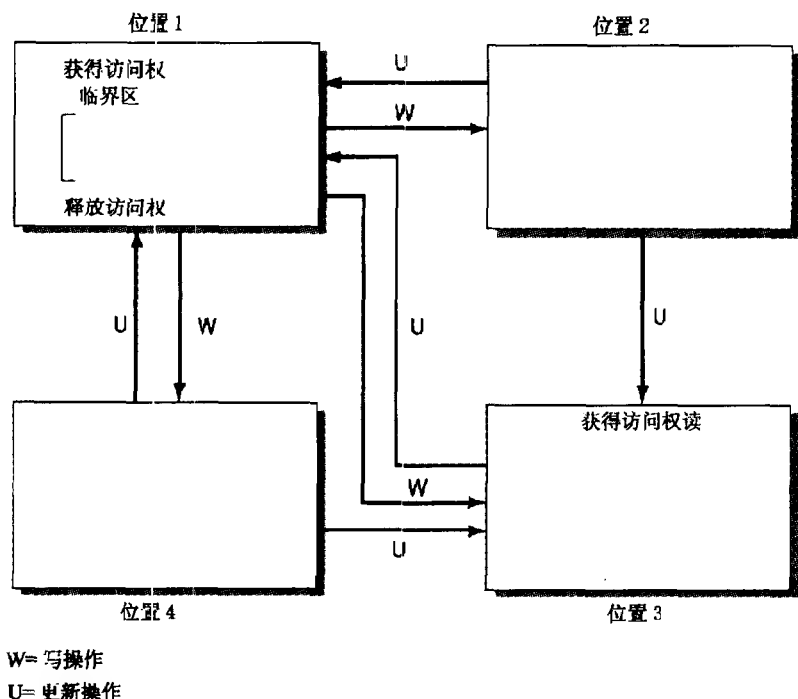


图 9-8 释放一致性中使用获得访问权和释放访问权

如果不利用临界区，释放一致性提供了**栅栏**（barrier）。栅栏通过区分执行的不同阶段来提供同步。在第 n 阶段中使用的栅栏在所有的进程都结束第 n 阶段之前不允许任何进程进入第 $n+1$ 阶段。如果你把整个的一个阶段作为临界区的话，应用带栅栏的获得和释放访问权的方法同临界区的实现实际是等效的。对应起来，进程进入一个新阶段时，它获得了访问权。当它到达栅栏的时候，它就要释放访问权。通常使用一个集中式的服务器来保证所有相关的进程都结束了给定的阶段。一旦服务器认定一个阶段已经结束了，它将通知所有处理器可以进入下一阶段了。

释放一致性使得指定一个变量保持一致性成为可能，尤其是它不需要所有的变量都保持一致性，只有那些指定为**保护性的**才需要。那些指定为保护性的变量或存储部分一定要遵从以下规定，这些规定是依照释放一致性模型建立的。

1. 一个进程访问共享变量之前，它之前所获得的所有访问权必须都已成功结束操作。
2. 释放访问权之前，必须结束所有读写操作。
3. 所有的获得访问权和释放访问权都必须遵从处理器一致性模型。

释放一致性允许相互无关的获得、释放访问权和栅栏各自独立地发挥作用。一个采用释放一致性的 DSM 系统可以获得和采用顺序一致性模型的 DSM 系统相同的结果。Gharachorloo 等人在 [GLLGGH90] 指出只要遵循这三个条件和释放一致性模型，正确使用获得访问权和释放访问权同步工具，就能获得这个结果。

9.3.8 懒释放一致性

懒释放一致性 (lazy release consistency) 模型是在 1992 年由 Keleher 等人提出的 [KCZ92]。基本的释放一致性模型相对于弱一致性模型确实提高了性能，但还可以再进一步提高。在释放一致性模型中，释放访问权是用来通知系统进程正在退出临界区，对共享内存所作的本地修改将传播到其他各个成员。在懒释放一致性模型中，这些修改不是马上进行传播的，而是只在要求传播时才传送到网络上。明确地说，对共享数据的改变只有在其他位置获得访问权的时候才被传达到。释放一致性的其他方面都保持不变。这一点变更提高了性能，尤其是在网络流量方面。如果一个地点不访问共享变量，变量的修改就不会通知到它，如图 9-9 所示，可与图 9-8 进行比较。

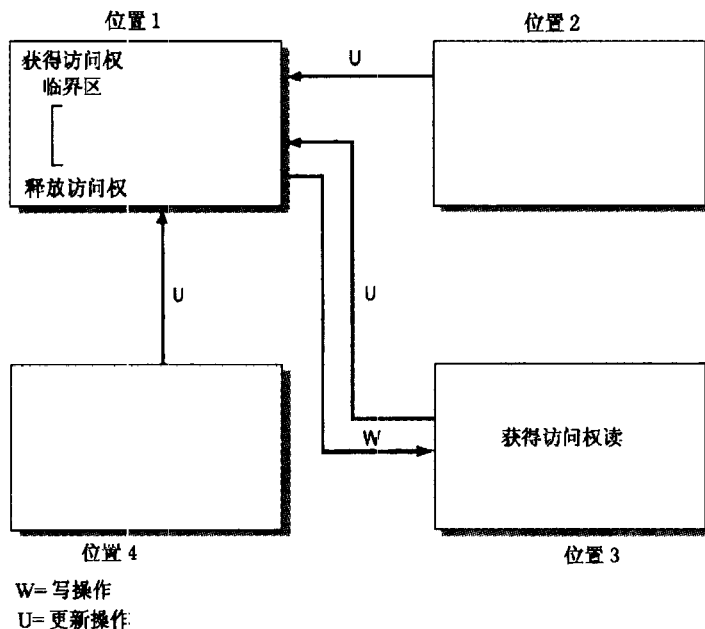


图 9-9 懒释放一致性

9.3.9 入口一致性模型

入口释放一致性 (entry release consistency) 模型是由 Bershad 等人提出的 [BeZe91, BZS93]。这个模型和基本的释放一致性模型有些类似。除了利用获得访问权和释放访问权的方法外，入口一致性模型要求每个变量与一种同步变量相关联，例如锁或者栅栏。这种相关

联的要求增加了程序员额外的负担，但同时也获得了更高的性能。如果这种方法能正确运用（就是运用适当的同步技术），共享内存的访问就能持续地保持一致。

9.4 两阶段提交协议

绝大多数事务依赖于 Gray 首先提出的两阶段提交协议（Two-Phase Commit Protocol）[Gra78]。其目的是保证在分布式系统中所有的事务能自动发生，在一个不变的状态下保持数据。这个协议包含了以下两个阶段。

1. 准备提交阶段。
2. 提交阶段。

一般来讲，第一阶段是为完成事务而进行协商和准备的阶段，第二阶段是依照协定执行事务的阶段。两个阶段如图 9-10 所示，我们现在分别进行讨论。

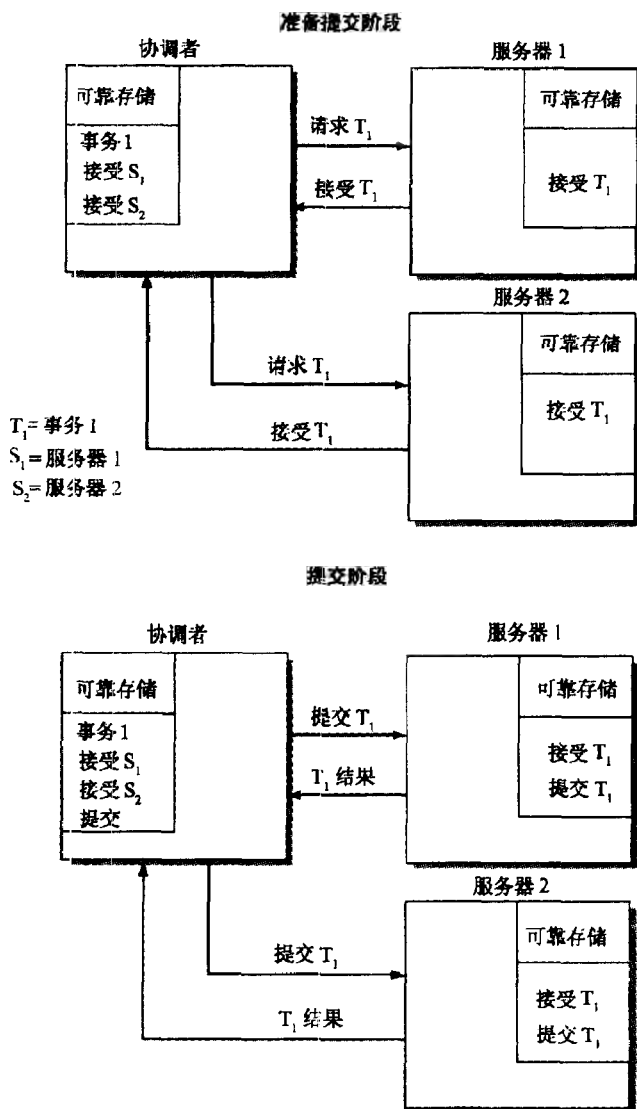


图 9-10 两阶段提交协议

9.4.1 准备提交阶段

两阶段提交协议为每个事务引入一个协调者，一般是发起地点。协调者负责征求和接收执行事务所必需的服务器的提交。以下是执行的步骤。

- ◆ 识别什么资源是必需的，要接触哪些地方的成员。 225
- ◆ 与所要求的场所接触，请求它们提交以完成动作。
- ◆ 记录每个所接触的服务器发回的回应。
- ◆ 如果一个服务器无回应或者做出否定的回应，试图征求其他服务器对提交的支持。 226

当一个服务器接收到一个请求，它必须决定它是否能够完成这个请求。一旦一个服务器提交了一件事务，它就不管怎样都必须完成此事务。一个事务请求的提交回应就像一个有约束力的合同。然而有一点要注意，一个服务器接收到一个请求并发送一个提交回应以示接受并不代表事务就会发生。具体地说，直到进入提交阶段，事务还只是计划中的事务。一个地点的成员可以在收到请求的时候就开始执行事务，但是如果事务在进入提交阶段前被取消，它就必须作好回滚的准备。回滚意味着在执行事务的任一阶段之前将所有数据全部返回其原有状态。所以，在完成事务的各个部分之前，必须记录下所有的原有状态。最后数据的结果是看上去事务任何部分没有执行过，数据也处在安全、原始、固定的状态。

关于进入提交阶段的问题仍然存在。为了进入提交阶段，事务协调者必须收到事务的所有方面的提交消息。并且仅在这一点上，事务协调者向每个提交地点发出一个提交消息，然后事务处理进入第二阶段——提交阶段。如果事务协调者没有收到事务所有方面的提交消息（可能是由于缺少应答），就不能进入第二阶段。这时事务协调者必须通知服务器，而服务器必须回滚所有为了完成它的那部分事务而执行的动作。这样，完成事务的失败企图就不会影响到系统的状态了。注意：只有协调者有权力放弃事务，一旦一个服务器返回了提交消息，它就不能改变决定了。

9.4.2 提交阶段

一旦提交消息被接收到，服务器就可以返回它们的执行结果，并认为所作的更改是永久的。所有的服务器都完成了各自的动作后，事务就完成了。听上去很简单，但是如果一个提交的服务器崩溃了怎么办？

为了让两阶段提交协议正常运作，它必须能承受系统故障。为了达到这一点，每个收到提交请求的服务器要把此请求记录在非易失性存储器中。这一步要在服务器向协调者返回消息之前完成，以防因为服务器在返回消息和记录请求之间发生的崩溃而导致的混乱。除此以外，从事务协调者发出的提交消息和协调者那里的事务状态也要记录在非易失性存储器中。这样，所有的成员，包括协调者，都存有一个关于它们对于事务状态的职责的可靠复制。当任何成员（包括协调者）从崩溃中恢复的时候，它检查非易失性存储器和这些防止崩溃的记录。如果有提交事务的记录，则它马上去完成那部分的事务工作；如果这个成员不是协调者并且有记录将要提交事务，但是没有从协调者发送来的确认信息，则它必须与协调者联系，以检查是否进入了提交阶段。 227

9.5 嵌套事务

事务可以是简单事务或者**复合事务** (compound transaction)。复合事务包含了许多**嵌套事务**，而简单事务则不是。复合事务中的每个嵌套事务也可以是一个包含了许多嵌套事务的复合事务。复合事务可以形象地用树来表示，原来的事务就是父结点，嵌套事务的第一层就是树中的下一层。包含在父结点中的事务有时也称为子事务。设计和组织带有嵌套子事务的事务有以下几点好处：

- ◆ 提高并发性。
- ◆ 提高性能。
- ◆ 提高容错性。

利用并发控制，处于同一层次的子事务可以同时执行。这种同时执行增加了并发执行的能力，同时也提高了性能。每一个子事务各自独立地提交和中止。所有子事务必须在父事务进入提交阶段前完成提交。如果一个子事务中途退出，父事务可以有几种方法来完成子事务，包括重新选择一个客户来执行。每一个子事务都有其子目标和父事务的检查点，因此整个层次的设计对事务管理很便捷。此外，每个子事务都具有并遵从所有的 ACID 特性。因此，子事务的结果只在结束的时候才对其父事务可见，每个子事务都看作一个原子操作。最后，如果父事务决定它无法进入提交阶段，事务和与其相关的嵌套事务必须回滚。详细说明

228

9.4 和 9.5 分别给出了嵌套事务的准备提交阶段和提交阶段的算法。

详细说明 9.4

嵌套事务的准备提交协议

```

Commit flag = 1
//初始的提交标志为 1，代表真
flag PREPARE-TO-COMMIT (transaction a)
//函数的返回值表示提交状态，
//事务作为输入参数。事务
//结构有责任维护子事务的信息

IF childless (transaction a)
//这个事务内不包含嵌套事务，
//不是复合事务

THEN
    flag = flag & Request commitment (transaction a)
//通过将 flag 现值和提交请求的
//返回值进行“位与”操作，
//flag 值只有在全部同意提交的
//情况下才为 1。有些实现在此
//处可能更加复杂以识别是谁不
//能提交，以便进一步作出决定
  
```

```

ELSE
    //这个事务含有子事务。准备
    //提交不是用作复合事务
    //的，所以我们必须递归
    {
        FORALL children (transaction a) DO
            //我们将一直对每个子结点或子
            //事务进行递归直至达到基本的
            //简单事务
        PREPARE-TO-COMMIT (transaction a (current child));
        //所有子事务的递归函数调用
    }
}

```

229

详细说明 9.5

嵌套事务的提交阶段协议

```

COMMIT-PHASE
IF (flag=1)
    //准备提交阶段成功，可以进入提交阶段
    THEN IF childless (transaction a)
        //利用嵌套结构的优点执行循环，直到没
        //有子结点的事务为止
        THEN
            record in stable storage
            (commit, transaction a)
            //一定要记录在非易失性节点中，事务已
            //经处于提交阶段而且消息正在向客户发
            //送
            notify commitment (transaction a);
            //通知客户提交阶段已经开始
        ELSE
            //当前的事务是复合事务
            {
                FORALL children (transaction a) DO
                    //进行循环直至达到基本的例子——仅
                    //包含一个简单的事务
                COMMIT-PHASE (transaction a (current child));
            }
        }
    }
}

```

```
//进入则结束。准备提交阶段。可以增  
//加有关如果不能进入提交阶段而采取  
//动作的其他条件，或者调整进入提交  
//阶段的栅栏
```

9.6 事务实现中的问题

230

虽然事务系统的实现看上去很简单，这一节还是要给出事务系统实现中会遇到的困难和问题。为了保证正确的事务，这些问题必须避免，同时必须遵守 ACID 特性。

9.6.1 预读写

在 9.1 节中，我们看到了不采用事务而导致的两个问题。不幸的是，如果利用没有正确实现的事务，同样会产生这样的问题。其他可能发生的有关问题包括了预读写。一个事务对一个数据进行读操作，而这个数据正在被其他将要退出的事务操作，这时预读的问题就发生了。如果一个事务对一个数据进行写操作，而这个数据正在被另一个事务操作，就产生了预写的问题。两个问题的发生都牵涉到独立事务的相互作用，事务管理服务采用延迟执行就可以避免这些问题的发生。延迟执行使得正在被事务操作的数据不能被其他事务读写，直到当前事务进入提交阶段或者中途退出。少量的延迟会稍微降低系统的性能，但是它保证了事务独立性的保持和加强。

9.6.2 中途退出的多米诺效应

如果一个事务不能进入提交阶段，它就有必要退出。如果其他事务看到了与退出的事务有关的结果，这个事务的退出可能遭受多米诺效应 (Domino Effect)。这种多米诺效应特别在事务管理系统遭到预写操作后可能发生。任何以预写的数据为执行基础的事务在预写的事务中途退出后也都必须退出。同样，任何利用第二代退出事务结果的事务都要退出。这种因中途退出而导致的结果是很惊人的。

9.6.3 保证恢复能力

下面的两个步骤将极大提高事务管理系统的恢复能力，我们主要的目标是保证 ACID 特性得以确认。

- ◆ 如果另一个未完成的提交请求牵涉到相同的数据，则应延迟当前的提交。这也有助于防止因退出而产生的多米诺效应。
- ◆ 利用数据的本地暂时拷贝直到正式进入提交阶段。暂时拷贝可以保存在本地的易失存储器中。如果一直不进入提交阶段，则删除暂时拷贝。如果进入提交阶段，则把暂时拷贝复制到永久的不易丢失的存储器中。这一步也有助于减少或消除预读写的发生，因为暂时拷贝是保存在本地的。

231

9.7 小结

在这一章中，我们学习了分布式系统中非常重要的一个题目——事务管理。事务管理用

来保证 DSM 和目录服务中数据操作的正常进行。要保证事务管理的成功,就需要在系统发生故障的时候能够回复并继续运作。也就是说,它必须具有容错性。在关于两阶段提交协议的讨论中,我们已经看到实现一个容错的事务管理系统是可能的。

表 9-1 一致性模型及其要求的小结

一致性模型	要 求
严格一致性	必须总是读取最近更新的值。依赖全局时间
顺序一致性	一起共享通用的全局视图。依赖全局次序
偶然一致性	偶然相关的事件必须以相同的顺序在视图中出现
PRAM 一致性	一个进程中的所有的写操作必须以相同的顺序出现在所有的系统中
处理器一致性	PRAM 一致性加上存储一致性。所有在给定位置的操作都必须在整个系统内有一致的顺序
弱一致性	利用同步变量来执行写操作和更新操作。同步变量的访问必须遵循顺序一致性。全部的读写操作必须在释放访问权前完成。最后,所有的得到和释放访问权必须遵循处理器一致性模型,无关数据的同步变量可以独立并行地运行
释放一致性模型	与弱一致性相同,但是把同步动作分为获得访问权和释放访问权。获得访问权在进入临界区时执行,并处理所有更新。释放访问权在临界区结束时执行,并分发所有修改结果。这个模型同时也支持栅栏的概念
懒释放一致性模型	除了只有在要求时才进行修改结果的传送外,与释放一致性相同
入口一致性模型	除了所有的同步变量都要与锁这样的同步变量相关联外,与释放一致性相同

如果不能保证数据的一致性,任何操作系统或者在操作系统中的事务管理服务都将是不胜任的。在这一章中,我们学习了数据一致性的确切涵义,还考察了大量通用的模型。表 9-1 是各模型以及它们相关要求的一个小结。数据一致性也许比较复杂,不熟悉分布式系统的开发者也许会错误地认为数据一致性是当然的。就像对一个集中式的程序员来讲,理解一种语言中的几种参数传递方式是很重要的一样,分布式环境中的开发者必须对所采用的数据一致性模型有相当透彻完整的理解。理解此模型有助于保证适当的开发、执行以及对环境的解释。现在有几种严格程度不同的模型,很多研究者发现那些不很严格的一致性模型不但很适合,而且效率很高。对需要考虑网络延迟的人来讲这是一个非常重要的方面。在不远的未来,发现胜任而且有效率的一致性模型并将其运用于事务管理服务中仍将是一个活跃的研究领域。

232

9.8 参考文献

第 1 章中列出的大多数参考资料都可以提供有关事务管理的一般信息。[Prim95] 中描述了 Tuxedo 在线事务处理系统。另外,以下的书籍可能提供其他一些有用的信息: [GrRe93, Jal94, PTM98, 以及 RaCh96]。一些典型的有关这一章信息的研究论文包括 [AdHi90, BeZe91, BZS93, DSB86, DSB88, GGH91, Goo89, GLLGGH90, Gra78, HaRe83, HuAh90, ISL96, KCZ92, Lam79, Lam81, LiHu89, LiSa88, ShSn88, Wei91, 以及 Wei93]。

以下是有关 Transarc 公司及其商业事务管理产品 Encina™ 信息的网站: www.transarc.com。你也可以在第 1 章中列出的网站中搜寻到有关信息。

习题

9.1 描述三个用到事务管理的日常实例。对每个实例讨论遇到更新遗失和恢复不一致

问题的后果。

- 9.2 对事务的 ACID 特性中的每一项，描述一个如果不遵守这项特性将会导致不正确的事务的场景。
- 9.3 如果不遵守事务的独立性，将会出现哪些危险和潜在的问题？
- 9.4 指出四个受分布式操作系统选择的一致性模型影响的专业领域，描述一致性模型的选择是如何影响每个领域的。
- 9.5 为以下每条说明描述一个例子。
- a. 一组遵循顺序一致性但是不遵循偶然一致性的动作。
 - b. 一组遵循偶然一致性但是不遵循 PRAM 一致性的动作。
 - c. 一组遵循处理器一致性但是不遵循 PRAM 一致性的动作。
- 9.6 描述以下一致性模型的优缺点。
- a. 弱一致性。
 - b. 释放一致性。
 - c. 懒释放一致性。
 - d. 入口一致性。
- 9.7 事务管理服务是怎样为整个分布式操作系统的容错性能做贡献的？举例说明。
- 9.8 在什么情况下，一个已经接受事务请求的客户会被迫回滚？描述一个文件服务可能采用的保证执行回滚的方法。
- 9.9 两阶段提交协议中，为什么不允许客户改变关于接受事务请求的决定？
- 9.10 描述由于允许事务包含嵌套事务而产生的三个优点和三个随之而来的问题。

第 10 章 分布式同步

同步需要全局时间或全局排序来实现。集中式系统中时间的概念实际上非常简单：所有进程都使用同一系统中的时钟。相反，分布式系统中的时间非常像在第一天上课时把教室内所有人的手表调整到同一时刻。如何使他们在期终考试时同步呢？在这一章中，我们将探究与全局时间及其替代方法——全局排序有关的问题。10.1 节给出了有关全局时间概念的总体介绍。10.2 节讨论有关物理时间的问题和算法。10.3 节描述了一个 Internet 上的物理时间标准——网络时间协议（NTP）。10.4 节讨论有关逻辑时间和逻辑排序的问题和算法。

10.1 全局时间介绍

一个分布式系统包含很多不同的地点和个别的计算机系统，每一个地点和系统都有其各自的本地时钟。即使有可能在某一时刻让所有的时钟同步，各处系统时钟也会**偏移**（drifting），这和我们手腕上的手表没什么不同。偏移就是因为各个计时机制中微小的不精确性而引起的，致使一度同步的时钟逐渐变得不一致。在我们对高级系统的研究中，已经看到了很多需要某种程度的全局时间（global time）或全局排序的协议。**全局时间**用来给进程和数据提供时间戳。如果一个成员的时钟走得慢了（比如说时钟是 9:15，而实际上是 9:25），那么所有在那个地点的进程将在整个系统中得到不公平的待遇。在时钟不正确的系统中，每个进程或动作将显得比实际情况要陈旧一些。

235

讨论同步时需要阐明的一个重要概念牵涉到相对性。计算机关心时间的方式与人类不同。与人类不同，系统只有在决定事件的相对次序的时候才关心时间。因此，讨论时间的时候，认识到有两种根本不同的时钟是很重要的。涉及“人类”时间的时钟叫做**物理时钟**，只涉及相对时间和保持逻辑一致性的时钟叫做**逻辑时钟**。我们现在来考察两种时钟各自的问题。

10.2 物理时钟

物理时钟用来在分布式系统中传递一定意义上的“人类”时间，它们是计算机系统的手表。“手表”服务是一种时间服务，可以通过集中方式或分布方式实现。它有责任为系统内的进程和服务提供某种程度的全局同步时间。全局时间的实现有两方面需要考虑。第一方面包括如何获取准确的物理时间值，这将在 10.2.1 节中加以讨论。第二个方面要对分布式系统中的物理时间加以同步，这将在 10.2.2 节中讨论。物理时间服务可以使用集中式的算法来实现，这将在 10.2.3 节中讨论，也可以使用 10.2.4 节中讨论的分布式算法实现。详细说明 10.1 描述了典型的计算机时钟在单个系统中是如何实现的。

10.2.1 获得准确的物理时间

为了获得准确的物理时间值，要求物理时间的服务器从世界标准时间协调（universal

236

time coordinator, UTC) 获取当前时间。UTC 是当今所有时钟基准的国际标准。UTC 的两个来源分别为位于科罗拉多 Ft. Collins 的 WWV^①短波广播电台和地球观测卫星 (Geostationary Operational Environmental Satellites, GEOS)。在可以用 UTC 时间同步以前, 时间服务必须先调整数值以适应传递时间中发生的延迟。一旦时间值的报告从 UTC 服务器处返回, 它立刻就过时了, 这是由于在向客户传送时间的过程中时间本身也发生了流逝, 如图 10-1 所示。必需的修正值因为网络状况和大气层状况的变化也不是一个常数。所以, 准确的延迟因而也开始变化, 而获得真正准确时间的过程就变得复杂了。可以给计算机配备一个**时间提供者** (time provider), 一个能够直接从 UTC 服务器获得信息并作出有关传递延迟的适当调整的商用设备。有的系统也开始使用 UTC 的俄罗斯“堂兄弟”——全球移动通信系统 (global system for mobile communications, GSM)。就像生活中的很多东西一样, 时间提供者的质量也就是时间的准确性因其价格而变化。一般来说, “廉价”的时间提供者模型要花费数千美元, 而绝大多数准确的时间提供者模型要花费数万美元。

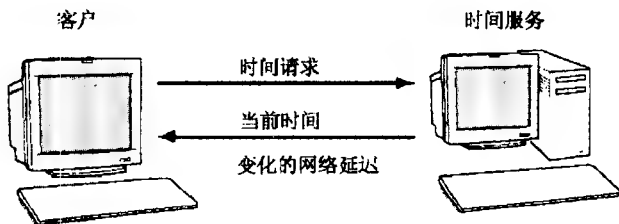


图 10-1 传递时间时网络的延迟

10.2.2 物理时间的同步

物理时间服务的第二个方面是分布式系统中时间的同步。在研究分布式系统中时钟是如何形成同步并保持同步之前, 首先定义同步的含义是很重要的。以前, 我们已经知道时钟受到固有的偏移问题的困扰, 这是不可避免的。因此假定分布式系统中的所有时钟总是精确地同步是不现实的, 这样就必须允许并定义一个可接受的时钟偏移限度。**时钟偏移** (clock skew) 定义为两个时钟之间因为偏移而产生的时间上的差异。分布式系统物理时钟服务定义了一个系统中所允许的时钟偏移的最大值。只要两个时钟之间的差值小于最大时钟偏移量, 就可以认为时间服务保持了同步。

那么怎样才能确定是否所有时钟之间的差别小于最大时钟偏移量呢? 为了分析任意两个分布式系统内的单独时钟之间的差值, 每个成员都必须能够读取其他成员的时钟值。如地点 A 正在读取地点 B 的时钟, 为了获取数值, 要进行如下步骤的操作:

1. 读取数值所需要的信息必须经网络传递至 B。
2. 必须读取 B 的时钟值。
3. B 的时钟值传递回地点 A。
4. 调整 B 的时钟值以反映在网络上传输所花费的时间。
5. B 的时钟值与 A 的时钟值相比较。

第 4 步很难准确地实现。网络上传递任何信息所花费的时间不断地变化, 这可能是由于系统和网络的负载以及网络上传递信息所经过的路径的不同。另外, 如果网络上发生错误, 信息将多次传递, 尝试的次数也是增加延迟的一个因素。

237

① WWV 不是字母缩写而是电台寻呼字母。

详细说明 10.1 计算机时钟的实现

以下是六个计算机时钟的有关组成部件。

1. 石英晶体。
2. 计数寄存器。
3. 常数寄存器。
4. 固定起始日期。
5. 固定起始时间。
6. 滴答计数器。

我们首先研究一下前三个组件的用途。像石英表中的晶体一样，石英晶体以事先预定的固定频率振荡。这些振荡由计数寄存器记录。顾名思义，常数寄存器存储了一个由石英晶体的振荡频率决定的并依赖于它的一个常数值。石英晶体每振荡一次，计数寄存器内的值都要减小 1。当计数寄存器内的数值为 0 的时候，就产生一个中断并返回到它的初始值。为计数寄存器选择的初始值是基于石英振荡的，所以每次中断都尽量接近一秒的第 1/60。因此计数寄存器产生的每次中断都等于是时钟的一次滴答。

后三个组件是用来计算实际“人类”时间的。每个系统时钟的功能是由固定起始日期和固定起始时间决定的。在 UNIX 系统中，固定起始日期是 1970 年 1 月 1 日，固定起始时间是 0000。如果系统没有关于内存的电池备份，每次系统重新启动时系统管理员都必须输入当前时间和日期。这时，系统计算从固定起始日期起已经发生过的滴答数，并把此值放入滴答计数器。计数寄存器每发出一个中断，滴答计数器就增加 1。根据滴答计数器与固定起始日期与固定起始时间，系统就能计算出实际的“人类”时间。所有石英并不是完全相同的，所选择的初始值也可能不像人们所希望的那样精确，计算机时钟每到一阶段就需要重新同步，以保持最大的准确度。特别地，基于石英振荡的计算机时钟在不到两周的时间内要偏移 1 秒多的时间。

238

10.2.3 集中式物理时间服务

分布式操作系统中的集中式物理时间服务可以设计成两种工作方式：基于广播的，或者是请求驱动的。从我们学过的集中式方法来看，集中式时间服务器是分布式系统中故障的关键点，同时服务器周围的流量会增加。规模也是利用集中式方法的一个限制因素，但对小规模的分式系统来讲，集中式的解决方案可能非常成功。我们现在研究这两种工作方式及其相应的问题。

基于广播的方式

在基于广播的集中式物理时间服务实现方式中，时间服务阶段性地向分布式系统中的成员广播当前时间。可以采用两种方法接收所广播的时间。第一种方法，各成员考虑了网络传输时间造成的延迟后，与接收到的广播时间相比较来估算它们的时钟，并随之调整自己的本地时钟，但它们不会将时钟设定为更早的时间。这种方法将导致以下两种情景。

- ◆ 如果给定成员的时钟比时间服务器的时间快，成员将放慢它的时钟以使得它逐渐接近

正确的时间。时钟不能拨回，因为反映当前时间的事件已经发生了。

- ◆ 如果成员的时钟慢于时间服务器的时间，该成员向前拨动时钟，如图 10-2 所示。替代的处理方法包括逐渐加快时钟。

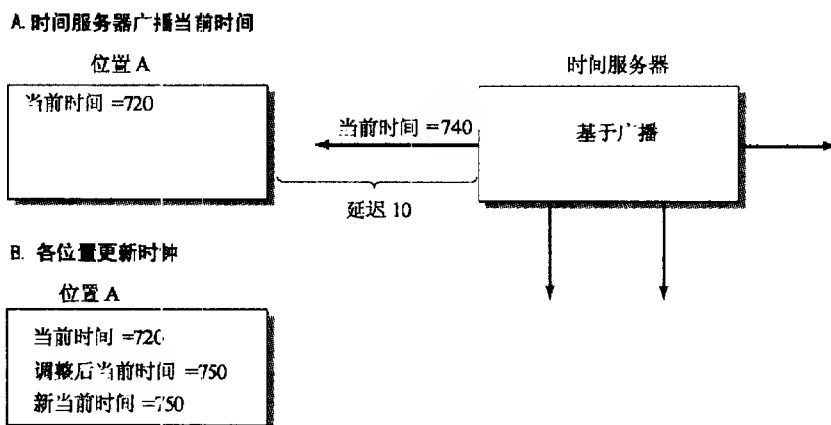


图 10-2 向前调整时钟

这种方法中，算法里没有考虑容错性。如果时间服务器的消息经历了比通常网络延迟更长的时间或者多次重发，成员将不会意识到这是不正确的。相反，成员将假定消息的传输看作是标准的一次传输，所经历的也是通常的网络延迟，并会因此调整自己的时钟。

基于广播的集中式物理时钟服务的第二种实现方式是伯克利（Berkeley）算法。伯克利算法由 Gusella 和 Zatti 提出 [GuZa89]。它之所以称为伯克利算法是因为它用于 Berkeley UNIX 4.3 系统中实现时钟的同步。它不需要时间服务器去访问 UTC。与第一种方法非常相似，集中式时钟服务器阶段性地广播当前时间。此算法包含以下步骤，如图 10-3 所示。

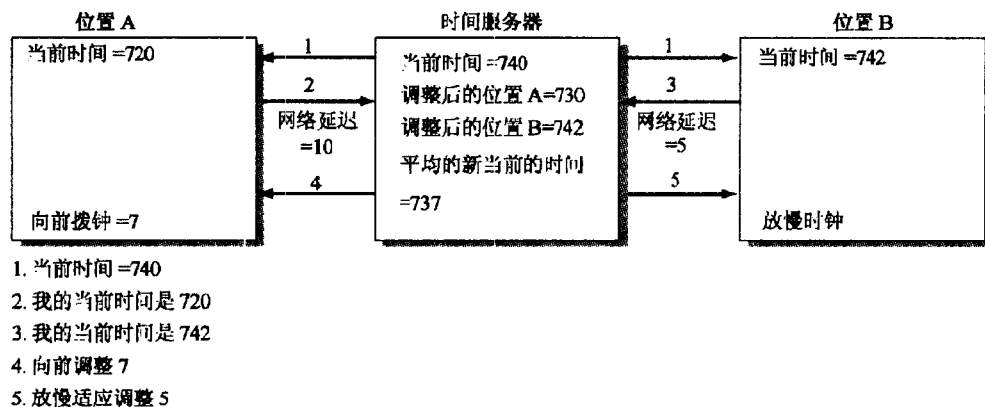


图 10-3 物理时钟同步的伯克利算法

1. 各成员接收到广播时间时，将各自的当前时间值发送到集中式时间服务器。
2. 时间服务器根据一个独立地反映各成员平均网络延迟的常数值调整从系统成员那里接收到的时间消息，该常数值是事先定义好的。
3. 调整后的时间如果和其他时间之间的差值超过了一个事先定义的常数，则它将被忽

略不计。将忽略的时间值认为是由于系统中的故障而导致的，并认为是不准确的。

4. 计算剩下的时间值同时间服务器中的时钟值的平均值，并认为这个平均值是当前时间。

5. 时间服务器计算各成员所需要的调整量，并将此调整量发送至各成员。

6. 各成员调整各自的时钟。同样，走得快的时钟不能回拨，所以这样的地点就需要拨慢它们的时钟。

240

请求驱动方式

请求驱动的方法最早由 Cristian 提出 [Cri89]。在集中式物理时间服务的请求驱动实现中，每个成员向时间服务器发出一个要求当前时间的请求。算法明确地包含以下步骤，如图 10-4 所示。

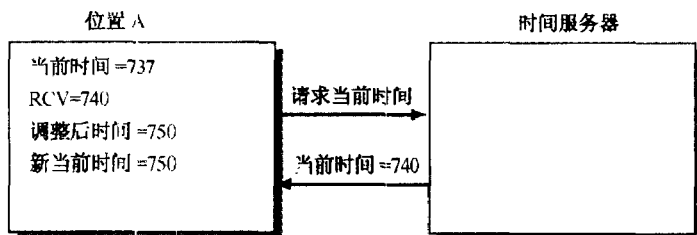


图 10-4 请求驱动的物理时间同步

1. 一个成员向时间服务器发出一个请求当前时间的消息。

2. 时间服务器返回它的当前时间值。

3. 成员计算它自己的时钟与从时间服务器返回的时间之间的差值，这差值就是调整时间。注意这种实现方法将经历网络延迟和中断延迟。中断延迟在时间服务中出现，包含了两个值：第一个值是要求时间服务器产生一个中断的平均时间，表示到达了一个消息；第二个值是服务器响应请求所需要的时间。

4. 如果这个调整时间大于预先定义的阈值，则认为它是不准确的，它的产生可能是因为额外的网络延迟的关系。不准确的值将被忽略不计。

5. 如果数值被认为是准确的，成员随之调整自己的时钟。同样，走得快的时钟不能回拨，所以这样的地点就需要拨慢它们的时钟。

241

10.2.4 分布式物理时间服务

实现物理时间服务的分布式方法要求每个位置的成员以预先定义的时间间隔广播各自的当前时间。因为时钟总是会偏移，所以不能指望广播的消息会精确地在同一时间发生。某一位置的成员广播了它的时间后，就启动了一个计时器，然后开始收集它所接收到的时间消息。每个到达的时间消息以本地的当前时间标记，这个过程持续到计时结束为止。计时结束的时候，每个消息均调整为反映了从消息源到本地的网络延迟时间的信息。此时成员以以下方式中的一种来计算时间的平均值。

- ◆ 计算所有消息的平均值。此平均值可作为当前时间。这种方法会导致错误的结果，因为有些消息可能因为发生了网络延迟或重发而不准确。
- ◆ 将每个值与错误估计阈值比较。错误估计阈值表示了一次单一传输中可能发生的最大

网络延迟。任何超出阈值的值将被认为是错误的并被抛弃。计算余下数值的平均值，此平均值可作为当前时间，如图 10-5 所示。

- ◆ 抛弃最高的 x 个值和最低的 x 个值，然后取平均值。这个方法有点像为体操或花样滑冰打分。其中，最高的 m 个值和最低的 m 个值被认为是错误的并被抛弃。计算余下数值的平均值，此平均值可作为当前时间，如图 10-6 所示。

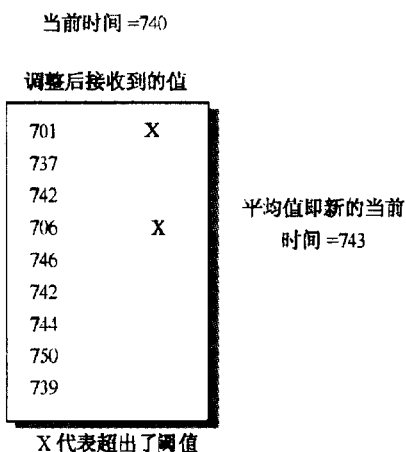
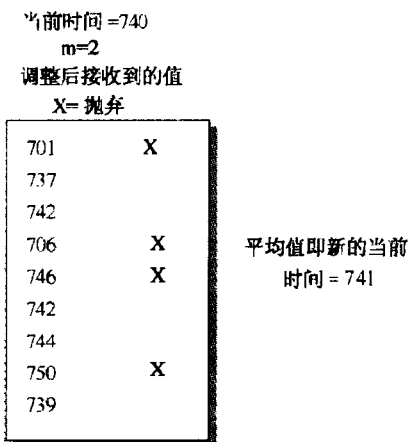


图 10-5 容错阈值方法

图 10-6 抛弃 m 个最高值和最低值

时钟此时以先前讨论的方式更新。从效率方面考虑，有些实现方法对广播的范围作了限制。要求所有的成员都要对其他的成员广播将造成很大的流量。所以，一种修改方案采取了要求成员只向其相邻的成员广播的方法。相邻关系要涵盖整个系统以提供一种全局的服务。这种修改大量减少了流量和每个成员计算新的当前时间所需接收的数值的个数。

242
243

10.3 网络时间协议

网络时间协议 (Network Time Protocol, NTP) 第 3 版的有关信息可以在 RFC 1305 [Mil92] 中找到。NTP 设计成在大型分布式系统中运行，允许系统间的连接从慢速的调制解调器直到可能的最快的连接。它广泛采用于 Internet。早在 1991 年，在北美、英国和挪威的 Internet 上已经有大约 30 个一级 NTP 服务器和超过 2000 个二级 NTP 服务器 [Mil91a]。第 3 版的出现更好地容纳了当今 Internet 所使用的高速网络。它向下兼容旧版本的 NTP，但极力推荐新版本以提供更高的准确性。它可以达到 1 到 50 微秒之内的准确性。尽管将它设计成在 Internet 的 IP 和 UDP 的顶层执行，但它也很容易被其他网络传输协议所采用，以致通常认为它是一个独立的协议。10.3.1 节定义了 NTP 的体系结构。10.3.2 节定义了 NTP 的目标。10.3.3 节描述了 NTP 服务模式。最后，10.3.4 节描述了简单网络时间协议 (Simple Network Time Protocol, SNTP)。

10.3.1 NTP 体系结构

为了适应 Internet 上巨大的数量的计算机系统，NTP 体系结构组织成层次状的树型结构。这个结构一定要是树状的，所以没有回路。整个体系结构中有许多棵树，每棵树的父结点都是一个一级服务器。一级服务器直接与 UTC 时间源相连接。NTP 的目的是将时间从这些一级

服务器传达到分布式系统中的二级成员。被二级服务器利用来获取时间信息的那个特定的一级服务器就是一级参考服务器。一级服务器可以有一个利用可靠性较差的方法获取 UTC 时间的后备服务器。后备服务器是出于容错性目的而设置的，只有当一级服务器崩溃时才发挥作用。二级服务器是以层次方式排列的。NTP 利用电话行业的术语来标注不同的层次。每一层次称为一层 (stratum)，如图 10-7 所示。父结点代表处于第 1 层的一级服务器。二级服务器处在第 2 层到第 n 层。第 2 层的成员直接与一级服务器相连接。第 3 层的成员与第 2 层的成员同步，依此类推。成员的层次数字越小，它所保持的时间就越准确，因为它与一级源头更近。准确性下降的范围与网络路径和本地时钟的稳定性密切相关。整个层次结构称为一个同步子网。

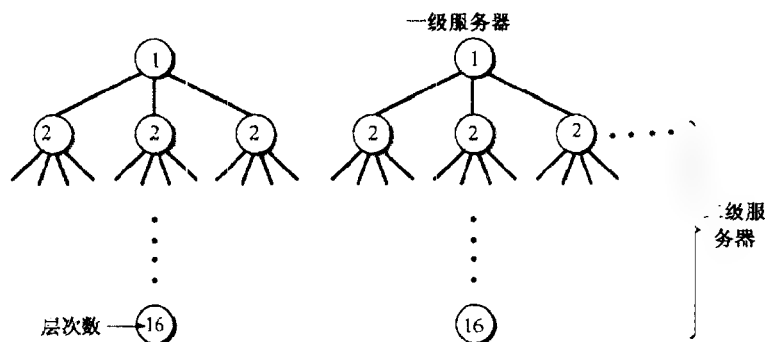


图 10-7 NTP 体系结构的层次

10.3.2 NTP 设计目标

以下是 NTP 的四个主要设计目标。

1. 允许准确的 UTC 同步。
2. 即使出现显著的连接上的问题也要保持正常运行。
3. 允许保持经常的重新同步。
4. 防止恶意的或意外的干扰。

为了达到第一个目标，NTP 必须能够为所有的客户提供服务，并在经常变动的网络延迟中提供准确的时间。这已经通过采用过滤数据的统计技术实现了。为了描述过滤技术，我们必须首先搞清楚 NTP 所传输的信息。NTP 设计为提供以下三部分与一级服务器有关的信息，数据的格式见详细说明 10-2。

244

详细说明 10.2 NTP 数据格式

NTP 时间戳以 64 位无符号定点数表示。所有时间戳均相对于 1900 年 1 月 1 日 0 时。时间戳的整数部分是前 32 位，时间戳的小数部分是后 32 位。64 位的数字足以表示直到 2036 年的时间。这种表示方法的精确度为 300 微微秒。选择这种格式允许多种不同精确度的时间表示方法可以方便地转换为时间协议的表示方法 (秒)，转换为其他时间协议也很方便。

协议中的所有数据以二进制补码表示，以定点数运算。可以指定数据为整数或定点数表示方法。利用定点数的表示方法时，不指定精度和小数点位置。所有数位从左边或者高位开始按照从小到大的次序排列。

1. 时钟偏移量。
2. 往返的延迟。
3. 离差值。

时钟偏移量定义为两个时钟之间的差值，尤其是指成员时钟为了与一级参考时钟同步而必须调整的值。往返的延迟使得成员能够在指定时间发出到达一级参考时钟的消息。离差值 (dispersion) 指本地时钟相对于一级参考时钟的最大误差值，离差值只能指定为正值。同时，这个信息还用来获得时间并指示时间的质量和准确性。离差值因子数值越高，数据就越不可靠，准确度就越低。所以，NTP 有能力区分不同时钟信息之间的准确率。

所采用的统计技术要求多个时间信息之间进行比较。时钟的同步需要在几秒钟内进行多次的交换。识别时钟偏移和在一毫秒内保持时间需要超过几个小时的大量的交换量。所以，统计的准确性与获得同步所需要的时间直接相关并且依赖于它。值得注意的是每一次交换，系统和网络的性能是相对没有影响的。

第二个目标要求 NTP 即使出现显著的连接上的问题也要保持正常运行。为了达到这样的目标，NTP 包含了冗余的时间服务器和服务器间的冗余路径。此外，在一个延长的时段内服务器仍没有到达时，允许进行简单的重新配置。重新配置包括对不可操作或发生错误的网络部分进行迂回的网络路由。所选择的新的一级服务器必须是可用的服务器中最近的一个。远近程度是由一级服务器与二级服务器间的网络延迟决定的，以此作为同步距离的参考。距离相同时，新的一级服务器就从几个距离相同的服务器中随机产生。如果某子网内所有的一级服务器都崩溃了，就启用后备服务器。

第三个目标要求允许保持经常的重新同步。这种重新同步对抵消计算机中的偏移率是必要的。允许经常更新的同时，NTP 也能够扩展它的服务以适应在大规模分布式系统中潜在的客户和服务。

最后一个目标包含防止恶意的或意外的干扰，这在大规模分布式系统中很重要。系统中的成员越多，系统中的时间服务意外地出错并发送错误信息的可能性就越大。另外，系统还可能包含恶意的时间服务。为了适应潜在的破坏，所有的时间服务都采用了身份认证（在第 11 章中讨论）并返回地址校验。另外，可以采用 64 位 DES 密钥（在第 11 章中讨论）。利用身份认证和加密技术以后，会遇到因为安全过程所需时间而产生的额外延迟。调整时间时必须考虑安全时间延迟和网络延迟的因素。像网络延迟一样，安全延迟不能准确地估量，这样调整后的时钟值就不会非常准确了。

10.3.3 NTP 同步模式

NTP 服务器可以以三种方式实现同步：组播，C/S 模式和对称模式。组播同步模式是用在高速局域网上的，它不是操作最准确的模式，但是对很多应用来讲经证明已经是足够的。组播 (multicast) 是发给一部分用户的网络消息。用组播模式操作时，时间服务器会周期性地广播当前时间。然后接收者利用这个时间来调整它们的时钟，把微小的网络延迟时间

考虑进去,如10.2.3节中描述的一样。这个算法类似于基于广播的集中式物理时钟同步的第一种方法。网络不支持组播的时候,大多数会利用C/S模式来实现操作。

C/S模式与10.2.3节中描述的请求驱动的方法有些类似。一个服务器接收系统中成员发出的请求,每一个请求接收到一个带有当前时间的响应。客户端在每一次重启动和此后的周期性间隔发出请求,间隔是基于客户的要求并由客户决定的,服务器不保留客户的状态信息。整个机制呈现为简单的远过程调用。

最后一个操作模式是对称模式。在要求最高级别的准确度时使用这种模式,它包含了一对交换时间信息的服务器。信息的交换是为了获得两个服务器之间的当前等待信息从而提高准确度。这种方法在层次数相差不超过1的时候表现最好。这种操作模式同C/S模式相比更能体现对等计算模式。

所有的操作模式都假定下层的传输协议(IP或者UDP)处理包括错误探测和重传在内的差错控制。详细说明10.3显示了NTP采用的帧格式。所有的操作模式在数据传送到时钟更新过程之前要执行八个确认测试。四个测试用来确认数据,四个测试用来确认附加在数据上的信息标题。详细说明10.4给出了所执行的八个测试的列表。

详细说明 10.3

NTP 帧格式

NTP帧格式遵守[Mil95]中描述的UDP帧格式。因为UDP是IP的一个客户,这个帧格式也服从IP协议。帧格式包含以下的数据域:

- ◆ LI (跳秒指示符)
- ◆ VN (版本号)
- ◆ 模式 (MODE)
- ◆ 层次 (Stratum)
- ◆ 轮询 (Poll)
- ◆ 精度 (Precision)
- ◆ 根延迟 (Root Delay)
- ◆ 根散布 (Root Dispersion)
- ◆ 参考标识符 (Reference Identifier)
- ◆ 参考时间戳 (Reference Timestamp)
- ◆ 原发时间戳 (Originate Timestamp)
- ◆ 接收时间戳 (Receive Timestamp)
- ◆ 传输时间戳 (Transmit Timestamp)
- ◆ 认证者 (Authenticator)

LI域即跳秒(leap)指示符,在帧中占两位,跳秒产生时需要标示。跳秒会不时地产生以保持基于原子钟的时间——UTC与天文时间之间的准确协调。这个数据域可能包含以下的数值和含义:

- ◆ 00: 无警告。
- ◆ 01: 上一分钟包含61秒。

◆ 10: 上一分钟包含 59 秒。

◆ 11: 报警状态: 系统未同步。

VN 域即版本号, 是一个三位的整数, 用来指示所采用的 NTP 版本号。如前所述, 当前的版本号是 3。

模式域是用来表明当前 NTP 操作所采用模式的三位数域。一共有八个可能的取值。模式 0、6 和 7 保留。模式 1 和 2 表示对称操作模式。模式 1 是基于广播的, 也称为主动对称模式。模式 2 是请求驱动的, 也称为被动对称模式。模式 3 表示客户模式。模式 4 表示服务器模式。最后, 模式 5 表示组播模式。

层次域有八位, 用来表明本地时钟的层次数。数值 0 未指定。如果层次数为 1, 本地时钟为像 UTC 广播时钟一样的一级参考时钟。如果层次数在 2 至 15 之间, 本地时钟就是像 NTP 服务器一样的二级参考时钟。网络时钟源大多层次数为 1 或 2。数值 16 至 255 保留。

轮询域是一个八位有符号整数, 用来表示连续的消息之间最大允许的时间间隔。所有数值都以与实际秒数最接近的二的幂表示。大多数应用程序使用数值 6 或者 10 分别表示 64 或 1024 秒。

精度也是一个八位有符号整数。它用与秒数最接近的二的幂来表示当地时钟的精度, 数值通常处于 -6 到 -20 之间 (工作站)。

根延迟是一个 32 位有符号的定点数。它用来表示一级参考的总回路延迟。时间还是以秒表示, 小数点位于第 15 位与第 16 位之间。数值可以为正也可以为负, 通常处于负若干毫秒到正几百毫秒之间。

根散布是一个 32 位无符号定点数, 用来表示有关一级参考的通常错误。数值以秒给出, 小数点位于第 15 位与第 16 位之间。数值经常在 0 到几百毫秒之间波动。

参考时钟标识符有 32 位。这个数值域用来标识信息源。其中内容可能表示一个 NTP 二级源的 IP 地址, 或者一个像前面讨论过的 WWV 那样的普通的广播服务, 或者像前面讨论过的 GEOS 那样的卫星服务。

所有的时间戳都有 64 位。参考时间戳表示本地时钟最后校正的时间戳, 起始时间戳表示请求的时间, 接收时间戳表示服务器接收到请求的时间, 传输时间戳表示服务器发出回答的时间。

认证者域是一个可选的 96 位数据域, 当选用认证选项时使用。认证将在第 11 章中讨论。

详细说明 10.4

NTP 的八个确认测试

以下是 NTP 在进入更新时钟过程前要执行的八个测试。前四个测试确保数据的有效性, 计算偏移量、延迟和散布需要合法的数据。后四个测试确保标题信息的有效性。这些测试决定是否一个对等者 (peer) 被选中用做同步操作。

1. 要求给出的时间戳不能和上次接收到的时间戳一样。这保证了这个消息不是一个

消息的副本。

2. 要求原发时间戳要与上次接收到的时间戳一样。这个测试保证了时间戳信息是按顺序处理的, 而且消息来自正确的一级服务器。

3. 要求原发时间戳与接收时间戳不为零。零时间戳代表时间服务器是不可达的或者没有达到同步。

4. 要求计算得到的网络延迟必须处于允许的范围之内。

5. 要求认证机制被禁止或者消息已经通过了认证测试。

6. 要求对等的时钟已经达到同步。

7. 保证时钟将不会与更高层次的时钟同步。

8. 要求一个合理的包延迟与散布值。

10.3.4 简单网络时间协议

简单网络时间协议 (SNTP) 是 NTP 的一个修改版本, 在不要求很高的性能时可采用 SNTP。SNTP 可以在所有 NTP 指定的模式中操作。这个协议实际上不是一个新的协议, 而是怎样简化对 NTP 服务器访问的说明。尽管是简化的, 它仍然可以在微秒级上保持准确性。有关简单网络时间协议 (SNTP) 的信息可以在 RFC 1769 [Mil95] 中找到。

这个协议的操作包括无状态的远过程调用。它严格要求处于 SNTP 客户位置的成员一定要处在子网中的最高层次。SNTP 客户不能用来与其他 NTP 或者 SNTP 的客户同步。尽管操作模式和数据格式都与 NTP 中的一样, SNTP 服务器不提供容错性和冗余性能的实现。因为缺少冗余性, 所以极力推荐 SNTP 服务器作为外部同步源的连接点, 例如一个可靠的 UTC 广播时钟。这种情况下, SNTP 服务器将处于第 1 层。因为 SNTP 不允许身份验证, 服务器必须为客户所知。

10.4 逻辑时钟

因为精确地校正时钟很难, 在分布式系统中利用物理时钟唯一地将事件排序也很困难。正因为如此, 人们思考是否可以用别的方法来将事件排序。在这一节中, 我们要学习在分布式系统中利用逻辑时钟来获得一个惟一的事件顺序。逻辑时钟的实质建立在 Lamport [Lam78] 提出的超前关系上, 我们在 10.4.1 节中研究这种关系。10.4.2 节研究这种关系在逻辑顺序中的运用。10.4.3 节中有一个使用逻辑时钟来获得总体排序的算法。

250

10.4.1 超前关系

Lamport [Lam78] 首先描述了超前关系 (Happen-Before Relationship)。如果事件 a 在事件 b 之前发生, 这样的关系表示为 $a > b$ 。在经典的论文中, 有如下关于事件次序的重要观察结果。

- ◆ 如果两个事件 a 和 b 在同一个进程中发生, 它们之间的次序是它们被观察到的次序, 则 $a > b$ 。
- ◆ 如果 a 向 b 发送了一个消息, 那么 $a > b$ 。也就是说, 不能在消息发送前接收到它。这个关系并不在乎事件 a 和 b 是在何处发生的, 这 and 第 9 章中描述的偶然关系有些类似。

- ◆ 超前关系具有传递性。如果 a 超前于 b , b 超前于 c , 那么 a 超前于 c 。也就是说, 如果 $a > b$ 且 $b > c$, 那么 $a > c$ 。

必须要注意到这个关系不是自反的, 一个事件不能超前于自身。任何不具备超前关系的一对事件都是并发的, 并发的事件没有顺序的先后。详细说明 10.5 给出了超前关系的一个例子。

详细说明 10.5 超前关系的例子

作为一个例子, 我们来考虑如图 10-8a 中所示的事件 a 、 b 、 c 和 d , 事件 a 和 b 是在同一地点发生的, 根据超前关系, $a > b$ 。事件 b 向另一个地点发送了消息。接受这个消息的是事件 c , 根据超前关系, $b > c$ 。根据传递性我们得出 $a > c$ 的结论。最后, 事件 d 在同一时间在另一个地点发生。因为事件 a 、 b 和 c 是在不同地点发生的, 和事件 d 没有关系。事件 d 就认为是和其他三个事件并发的。事件 d 和事件 a 、 b 、 c 之间没有次序上的关系。

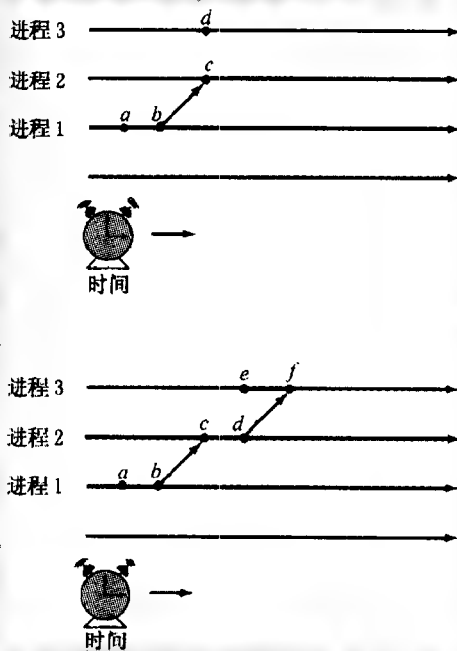


图 10-8 超前关系举例

为了进一步说明问题, 我们来看一下图 10-8b。这种情况下, 事件 a 、 b 和 c 保持了和第一个例子中相同的关系。然而, 事件 c 和 d 在现在的超前关系中是事件 d 和 e 。因为 $c > d$ 而且 $d > f$, 根据传递性就得到 $c > f$ 。最后, 因为事件 e 和 f 是处于同一个进程内的, 所以 $e > f$ 。要注意到事件 a 、 b 、 c 和 d 都是与事件 e 并行的。

10.4.2 逻辑顺序

逻辑顺序并不依赖普通的时钟, 普通的时钟无法存在于分布式系统中; 也不依赖于同步的物理时钟, 它们很难达到和保持。超前关系的优点是它不依赖于物理时钟。为了获得逻辑

顺序, 采用了与物理时钟独立的时间戳。时间戳可以由简单的计数器实现, 只要它们服从超前关系。每个事件都有一个时间戳。只要观察到的现象遵循超前关系, 就认为系统内的时钟是准确的。如果把事件 a 的时间戳记作 $T(a)$, 利用逻辑顺序的分布式系统必须支持以下的关系。

- ◆ 如果两个事件 a 和 b 在同一个进程内发生。它们之间的次序就是它们被观察到的次序, 也就是 $T(a) > T(b)$ 。给定进程的时钟不能回拨。
- ◆ 如果 a 向 b 发送了一个消息, 那么 $T(a) > T(b)$ 。也就是说, 不能在消息发送前接收到它。
- ◆ 如果 a 超前于 b , b 超前于 c , $T(a) > T(b)$, $T(b) > T(c)$, 那么 $T(a) > T(c)$ 。

Lamport [Lam78] 提出的算法融合了超前关系和按照以下两条规则实现的逻辑顺序。

1. 每个进程在每两个连续的事件之间增加时钟计数器的计数。这个时间戳反映了当前时钟的数值。

2. 如果 a 向 b 发送一个消息, 这个消息必须包含 $T(a)$ 。一旦接收到 a 和 $T(a)$, 接收的进程必须将它的时钟设置为 $[T(a) + 1, \text{当前时钟}]$ 中数值较大的一个。也就是说, 如果接收者的时钟落后了, 它必须向前调整以保持超前关系。

图 10-9 中所示的例子利用了详细说明 10.5 中所举的相同的例子, 在这种情况下, 使用计数器来获得逻辑顺序。有如下的超前关系: $a > b$ 、 $b > c$ 、 $c > d$ 、 $d > f$ 和 $e > f$ 。事件 b 携带了一个等于 2 的时间戳。所以根据规则 2, 进程 2 不得不在事件 c 发生的时候调整时钟。同样, 进程 3 不得不在事件 f 发生的时候调整时钟以保持超前关系。事件 e 是与事件 a 、 b 、 c 和 d 并行的。

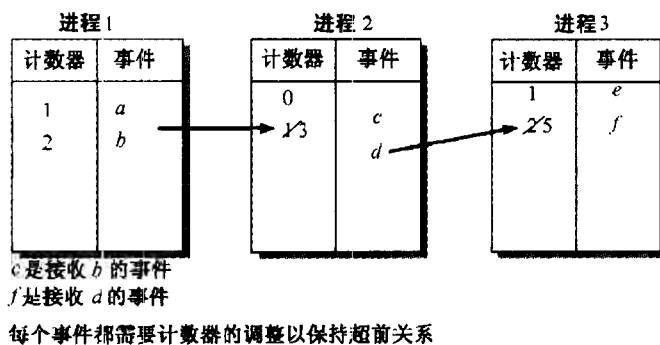


图 10-9 使用计数器的事件的逻辑顺序

利用物理时钟也可以实现逻辑顺序。一般来讲遵守逻辑顺序的第一条规则都不是问题, 计算机时钟在两个事件之间大多会产生若干次滴答。第二条规则也相当直观。如果到达的信息所携带的时间戳小于接收者的当前物理时钟, 那么物理时钟必须调整至此时间戳的下一个时间单位。必须牢记如果一个成员的时钟比正确的时间超前许多, 这个对物理时钟的负面影响将传播到整个分布式系统中。除了这个可能需要的回拨以外, 这种实现方法能有效地保持利用物理时间的事件之间的逻辑顺序。

图 10-10 中描述了一个利用物理时钟保持逻辑顺序的例子。又是同样的超前关系: $a > b$ 、 $b > c$ 、 $c > d$ 、 $d > f$ 和 $e > f$, 事件 b 携带了一个等于 20 的物理时间戳。所以根据规则 2,

进程 2 不得不在事件 c 发生的时候调整物理时钟为 21。同样，进程 3 不得不在事件 f 发生的时候调整物理时钟为 41 以保持超前关系。事件 e 是与事件 a 、 b 、 c 和 d 并行的。

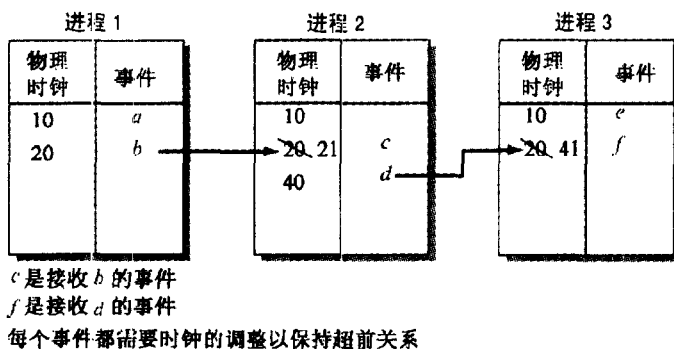


图 10-10 使用物理时钟的事件的逻辑顺序

10.4.3 带有逻辑时钟的总体排序

我们已经看过 10.4.2 中的例子，超前关系只能达到部分排序的程度。这种关系不能给出并发事件的顺序。为了获得**总体排序**，不允许两个事件可以精确地同时发生。首先由 Lamport [Lam78] 提出的一个直接的解决方案是利用在时间戳的低位部分填入进程标识号。因为每个进程均有一个惟一的进程号，就算在分布式系统中有两个成员分别在同时发生了两个事件，它们也持有独一无二的时间戳。例如，如果进程 001 与进程 002 都在时间 32 发生了一个事件，第一个进程的事件的时间戳是 32.001，第二个进程的事件的时间戳是 32.002。现在每个事件就能够拥有自己的独一无二的时间戳，系统也就可以获得总体排序了。

10.5 小结

在这一章中，我们把注意力集中在获得系统范围的同步所产生的问题及其解决方法上。同步有两个基础：反映“人类”时间的物理时钟和逻辑时钟的使用。如我们所见，物理时钟的同步是相当困难的。每个位置的时钟都会发生偏移，由此导致了它和系统中其他时钟之间的时钟偏移。无论是采用集中式的还是分布式的方法来达到同步，利用消息机制都会带来很多复杂的问题。它必须适应未知的同时也是不可预测的网络延迟。另外，如果采用请求驱动的方法，还要考虑到未知的中断延迟。如果没有关于从服务器的时钟上读取时间值之后已经经过了多少时间的精确信息，就很难去正确地调整时间，所以很多情况下采用了平均结果的方法。其实，最好的方法利用了“容错”的取平均值和抛弃那些看上去不合理的时间戳。我们也研究了一个广泛应用于同步物理时钟的协议——NTP，这个协议也允许对时间服务器进行身份认证。这当然是一个优点，但是它可能也加入了另一个不可知的因素，即认证消息所花费的时间。拥有如此多的变量，NTP 还能够在 300 微微秒内同步于物理时钟是很令人吃惊的。一般来说，这个精确度的级别即使对最复杂的应用也足够了。为了达到这样的精确度，要经过几个小时的消息和回应的交换。要快速获得准确时间的最好方法是直接使用 UTC 服务器，但这将带来大量的财政支出。

虽然利用物理时钟能够成功地实现同步，但它也会在分布式系统中带来不良的结果，许多应用要求的是总体排序。获得总体排序的最好也是最常采用的方法是基于 Lamport 的超前

关系的，并且这个方法也是由利用因果关系的逻辑时钟引出的。利用这种方法与物理时钟和进程 ID 相结合的时间戳，就能获得在分布式系统中的总体排序。

10.6 参考文献

第 1 章中列出的许多参考资料都可以提供有关分布式同步的大量信息。一些相当传统的有关这一章信息的研究论文包括 [Bra80, Cri89, GuZa89, HSSD84, KoOc87, LaMe85, Li-Ka80, LuLy84, MaOw85, Mil88, Mil90, Mil91a, Mil91b, Mil92, Mil95, Mit80, 以及 Ric88]。

以下提供了有关同步的一些 Internet 上的重要资源的链接，NTP 的 RFC 在 <http://www.cis.ohio-state.edu/rfc/rfc1769.txt>。SNTP 的 RFC 同时也包括 NTP 的声明信息，它的地址是 <http://www.cis.ohio-state.edu/rfc/rfc1769.txt>。所有与 NTP 有关的 RFC 可以在 <http://www.cis.ohio-state.edu/cgi-bin/waisrfc.pl> 找到。

习题

- 10.1 列举并讨论三个在分布式操作系统中试图与物理时钟同步时会遇到的困难。
- 10.2 列举三个在分布式操作系统中需要全局时间或全局次序的算法。
- 10.3 描述三种可能发生的难以获得和准确计算物理时钟值的事件。
- 10.4 在一个分布式系统中，绝大多数的计算机每毫秒滴答 1200 次，但是有一台计算机每毫秒滴答 1100 次，还有一台计算机每毫秒滴答 1230 次。计算系统中每分钟的最大偏移。
- 10.5 一个计算机的时钟是 11:07:34（时间格式是小时:分钟:秒），然而通知它当前时间为 11:07:26，计算它对自己时钟的调整量。应考虑到所做的调整必须使时钟在随后 4 秒内达到同步，请描述此调整方法。
- 10.6 命名并描述可以加入分布式时钟同步算法的三个容错特征。
- 10.7 使用分布式时间服务器的时候，利用网络邻居有什么好处？利用这种方法要注意些什么？
- 10.8 为什么 NTP 中层次数较高的成员的准确性较低？
- 10.9 为什么一个二级成员与一个一级参考服务器采用 NTP 的同步时钟时需要多次时间消息的交换？
- 10.10 描述在使用 NTP 作为时间服务系统中促进认证机制的使用的三个场景，并描述使用认证机制的缺点。
- 10.11 对以下每一个 NTP 的时间戳，描述其在分布式系统中的作用和优点。
 - A. 参考时间戳。
 - B. 原发时间戳。
 - C. 接收时间戳。
 - D. 传输时间戳。
- 10.12 为什么不要一个 SNTP 客户与另一个 SNTP 客户进行时间同步？
- 10.13 使用逻辑时钟的超前关系在以下的表格中填入校正后的时间（如需要），并标记出所有的超前关系以及所有的并发事件。

255

256

进程 1		
时间	校正时间	事件
2		A
4		B (向进程 2 发送消息)
6		C (从进程 2 接收消息 F)
8		D (向进程 3 发送消息)
进程 2		
时间	校正时间	事件
3		E (从进程 1 接收消息 B)
6		F (向进程 1 发送消息)
9		G (从进程 3 接收消息 J)
12		H (向进程 3 发送消息)
进程 3		
时间	校正时间	事件
4		I
8		J (向进程 2 发送消息)
12		K (从进程 2 接收消息 H)
16		L (从进程 1 接收消息 D)

257

10.14 写一个程序来实现逻辑顺序的超前关系。利用习题 10.13 作为一个示例的输入文件。要求程序必须能够计算每个进程的校正时间，并且标记出所有的超前关系以及所有的并发事件。

258

第 11 章 分布式安全

无论什么时候讨论分布式计算，首先提出的问题之一是：“它安全吗？”计算机安全经常包含两个部分：身份认证和访问控制。**身份认证**（authentication）还包含对合法用户的验证和身份鉴定。**访问控制**（access control）致力于防止对数据文件和系统资源的篡改。对一个独立的、集中式的单用户系统，例如 PC，只要把放置计算机的房间和磁盘都锁起来就能完成保卫工作。这样，只有拥有房间和磁盘钥匙的用户才能访问计算机的资源 and 文件。它同时完成了身份认证和访问控制。安全的程度就要看那把锁住计算机和房间的钥匙了。

对一个多用户的独立的集中式系统来讲，安全就稍复杂一些了。在这种情况下，身份认证包含对合法用户的验证和身份鉴定，通常通过某种形式的口令操作将身份鉴定包含在一起。为防止对文件、数据和资源进行篡改的访问控制通常通过访问列表来实现，并经常成为操作系统中的一部分。

259

在这一章中，同以前一样，身份认证和访问控制又一次成为将自己的系统与别的系统通过网络连接时首先考虑的问题。尽管身份认证和访问控制是作为操作系统操作的任务来考虑的，但是分布式访问控制可以由路由器和应用程序来执行。此外，既然是通过一个外界的资源（网络）来实现通信的，就有必要分别使用加密和数字签名来保护内容的安全，检验网上信息发送者的身份。

虽然分布式安全仍然是一个相当活跃的研究领域，当前的技术足以理解安全问题的基础，并且现在有很多针对分布式操作系统安全问题的解决方法。在 11.1 节中我们将研究多种的加密技术，包括它们在数字签名中的应用。在 11.2 节中展示了两种通常的身份认证方法。在 11.3 节中介绍在分布式系统中访问控制的多种方法。这一章中讨论的方法是除了集中式系统中必须应用的身份认证和访问控制之外而运用的方法，后者已在第 8 章中讨论过其中的一部分。

11.1 加密和数字签名

虽然计算机用户偶尔会想到用加密的方法来对自己独立的计算机系统上的信息进行安全保护，而更多的用户要求的是对于他们网络上的通信进行加密保护。**加密**（encryption）使用密钥将数据编码，这样窃听者就不会很容易地读出信息。加密后的数据称为**密文**（ciphertext），而加密前的信息是**明文**（plaintext）。从密文转换成明文的过程称为**解密**（decryption）。图 11-1 描述了一个典型的加密过程。Alice 和 Bob 使用凯撒码（the Caesar Cipher）对通信进行加密。凯撒码是一种轮转数据的简单加密体制，在此例中为向前轮转两个字母。显然，这不是一个保险的加密方法，可以经由一个类似谷物箱的解码环（decoder ring）进行解码。最通常衡量加密算法安全性的方式是**计算安全**（computationally secure）。如果不能通过现有的资源进行系统的分析来破解系统的话，加密算法就是计算安全的。加密分成两大类：私钥加密和公钥加密。除了对整体信息进行加密，两种方法都支持对文件进行数字签名。11.1.1 节以 DES 为例研究对称加密。11.1.2 节以 RSA 为例研究非对称加密。两个例子如果使用足够长

260 的密钥的话都是计算安全的。

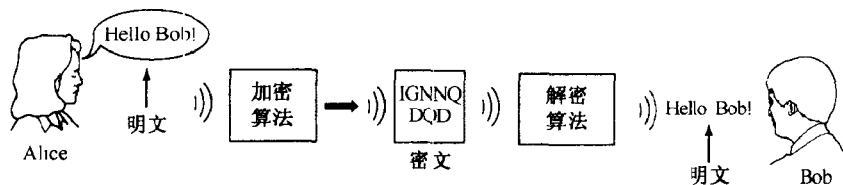


图 11-1 Alice 和 Bob 使用加密

11.1.1 对称加密

对称加密 (symmetric encryption) 指的是加密和解密使用同样密钥的算法。如：

$$E(p, k) = C \text{ 且 } D(C, k) = p$$

其中：

E = 加密算法

D = 解密算法

p = 明文 (原数据)

k = 加密密钥

C = 密文

既然加密和解密数据使用的是同样的密钥，这个密钥就必须秘密保存。这种加密方法也称为私钥加密或是传统加密。显然，使用这种系统的一个困难之处是传递密钥！解决这个密钥交换难题的一个简单的方法是先使用公钥加密的方法来交换密钥，然后再使用私钥加密。详细说明 11.4 描述了一个交换密钥的公钥加密算法，即 Diffie-Hellman 密钥交换算法。它非常实用，因为典型的 DES 加密差不多可以达到 45 000 kbps，而公钥加密一般为 20 kbps。我们现在来研究最流行的私钥加密算法——DES (读作 DEZ)，它是在 1977 年由国家标准化与技术研究所 (NIST)[⊖] 提出并作为美国标准的 [NIST77]。其他的私钥加密算法包括 IDEA [Lai92] 和 Skipjack (用在截割芯片中) [NIST94a]。

数据加密标准 (Data Encryption Standard, DES)

261 DES 在 1977 年作为一个标准实现，此后每 5 年重新审议，通常是在 12 月。所有美国的联邦办事处和其他代表它们处理信息的组织都必须使用 DES (用在未分类的文档)，它也普遍应用于非政府的团体中。它建立在使用 128 位密钥的 IBM 的 LUCIFER 系统上。一般来讲，密钥的数位越多，系统就越安全。DES 使用 64 位的密钥，其中 8 位是用做检错的，所以实际上 DES 有 56 位的密钥是用做安全目的的。因为它将二进制数据每 64 位作为一组进行加密，所以也称为分组加密。DES 的安全性是建立在密钥的安全上的，而不是算法的安全上。这种安全性通过密钥的尺寸得以加强，因为有 7 万兆兆 (70 000 000 000 000) 可能的密钥，所以，获得密钥的机会对绝大多数分布式系统来讲足够低了。当然，随着个人电脑平均性能的不断增长，顺序寻找密钥从而破解密文的能力也在相应增长。加密算法有三个阶段，如图 11-2 所示。解密算法就是将这三个阶段次序颠倒并在阶段 2 逆序使用分组密钥 (K_{16} 到 K_1)。

⊖ 从前称为国家标准化局，NBS。

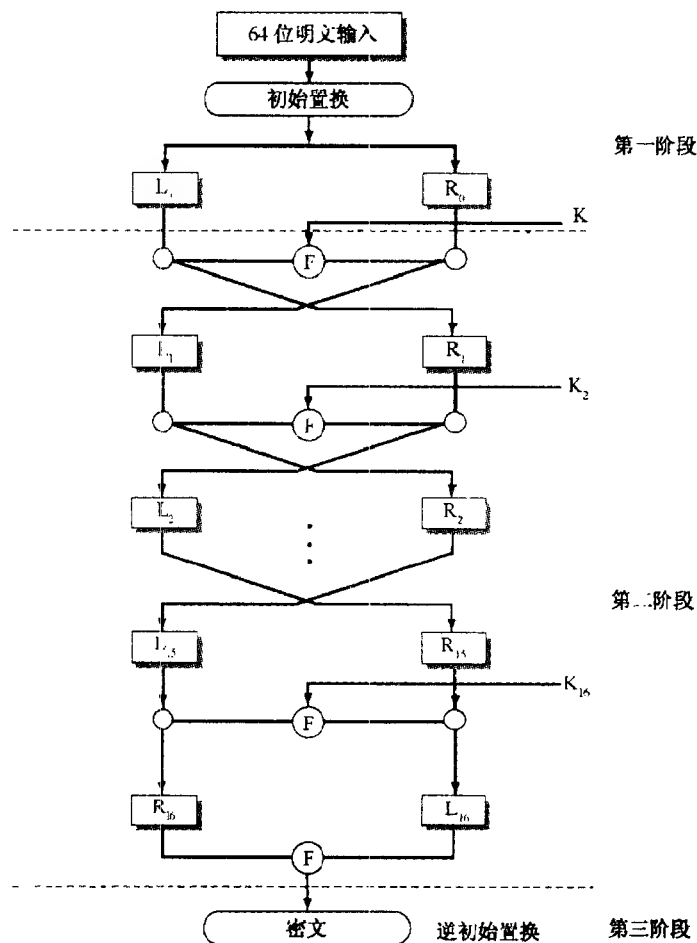


图 11-2 DES 的三个阶段

DES 第一阶段：初始置换

DES 的第一阶段包含一次 64 位分组的置换，将每个分组内部的位序打乱。**置换** (permutation) 这一术语有严格的数学意义，仅指改变次序。实际上置换是由一个表实现的（参见详细说明 11.1）。现在 64 位的数据分成两部分： L_0 （左段）和 R_0 （右段）。零下标表示原数据段。以后每经过一次 DES 算法中第二阶段的迭代，下标就增加。

详细说明 11.1

DES 置换

DES 标准所使用的初始置换的表如下表 11-1 [NIST77] 所示。所以，置换后的第 1 位是置换前的第 58 位，置换后的第 2 位是置换前的第 50 位，置换后数据的最后一位原来在明文中是第 7 位。

表 11-1 DES 初始置换 [NIST77]

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6

(续)

64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

DES 第二阶段：移位（重复 16 次）

第二阶段包含一个使用密钥的依靠表的算法。这个动作通常称为数据移位。这个算法要重复 16 次，因为每次使用不同分组的子密钥所以每次移位的动作都不一样。子密钥由另一组表和自己的移位算法决定。每次迭代后，L（左边）和 R（右边）的下标要增加以表示各个阶段，如图 11-2 所示。第 16 次迭代的结果称为预输出并传递给第三阶段。确切的表和各种算法见 [NIST77]。

263

DES 第三阶段：逆置换

DES 的最后一个阶段就像第一阶段一样，包含一次 64 位分组的置换，改变每个分组内部的位序，但是它使用的是不同的表。实际的置换是由一个表实现的（见详细说明 11.2）。这次置换的输出就是密文。

详细说明 11.2

DES 逆置换

表 11.2 所示的是 DES 标准进行逆置换时所使用的表 [NIST77]。所以，置换后的第 1 位是预输出中的第 40 位，置换后的第 2 位是预输出中的第 8 位，置换后密文的最后一位是预输出中的第 25 位。

表 11-2 DES 逆置换 [NIST77]

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

三重 DES

可以通过三重 DES 来加强 DES 的安全性。三重 DES 利用三个 64 位的密钥。数据首先通过使用第一个密钥的 DES 三个阶段加密产生 C_1 ，然后 C_1 通过使用第二个密钥的 DES 三个阶段加密产生 C_2 ，第二个密文 C_2 最后再通过使用第三个密钥的 DES 加密，如图 11-3 所示。

$$\begin{aligned} E(p, k_1) &= C_1 \\ E(C_1, k_2) &= C_2 \\ E(C_2, k_3) &= C_3 \end{aligned}$$

264

E 就是 DES 加密算法， k_i 是第 i 个密钥， p 是原来的明文， C_3 是最后的密文。

使用对称密钥加密的数字签名

在网络上传输数据时, 有两个基本方法来对文件进行数字签名。我们现在来研究第一个方法, 即使使用私钥加密的数字签名。数字签名也称为**消息摘要**, 并使用一个如[NIST93B]所描述的安全 hash 函数。这个 hash 函数称为**摘要函数**,

典型的有 128 位。这个摘要函数应用于整个文档, 产生一个依赖于消息中每一个比特信息的值。利用共享的私钥可以有两种方法计算摘要。最简单最快捷的方法计算消息的 hash 值, 再用私钥加密消息, 然后与加密后的摘要一同发送。接收者这时可以计算消息摘要, 加密并比较相互的值。如果两者相符, 说明文档的内容没有被改变。第二个方法是将密钥添加在消息后面, 然后计算 hash 值, 这种方法的结果如下所示:

计算 $D(M, K)$, 其中 D 是一个摘要函数, M 是消息, K 是共享的私钥。

文档现在可以公布或分发了。这个消息摘要另外有一个好处, 就是防止伪造自身的摘要值, 因为第三方不知道私钥。而私钥对计算正确的摘要值是必须的。无论哪种方法, 要证实它的完整性, 一定要知道密钥, 而且任何伪造的文档都会马上被发现。

11.1.2 非对称加密

非对称加密 (Asymmetric Encryption) 包含两个密钥——一个公钥和一个私钥, 也称为公钥加密。如果一段信息由公钥加密, 可以用对应的私钥以如下的方式进行解密。

$$E(p, k_u) = C \text{ \& } D(C, k_r) = p$$

265

其中:

E = 加密算法

D = 解密算法

p = 明文 (原数据)

k_u = 公钥

k_r = 私钥

C = 密文

如果一段信息由私钥加密, 可以用对应的公钥以下面的方式进行解密。

$$E(p, k_r) = C \text{ \& } D(C, k_u) = p$$

其中:

E = 加密算法

D = 解密算法

p = 明文 (原数据)

k_u = 公钥

k_r = 私钥

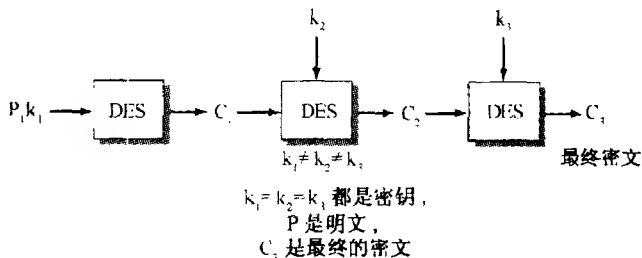


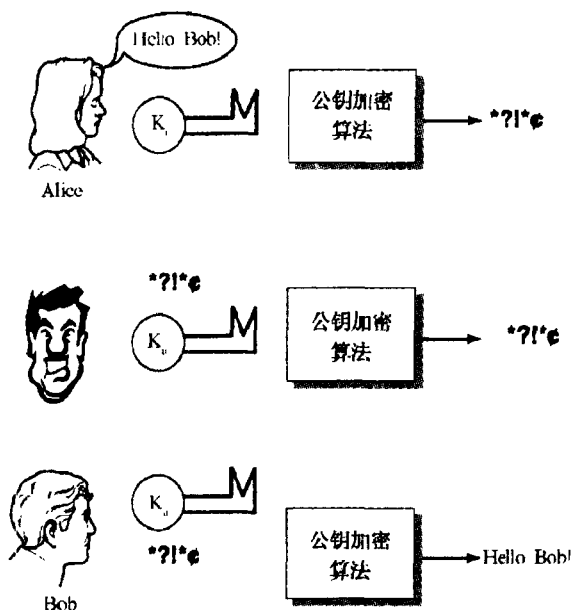
图 11-3 三重 DES

C = 密文

不能用加密消息的密钥对其进行解密，如图 11-4 所示。此外，从数学的角度讲，即使知道了其中一个，也很难获得另外一个。私钥应当由用户秘密地保存，就像它的名字一样。当然，如果每个人都知道了这个私钥，它就不再那么秘密了！公钥不必秘密地保存，可以通过公众列表服务向大家公布，通常使用 X.509 实现（将在 11.2.2 节中描述）。公钥加密的思想最早是由 Diffie 和 Hellman 在 1976 年的 [DH76] 的提出，背景是密钥交换的方法，详细说明 11.4 中有所论述。公钥加密最流行的形式是 RSA [RSA78]，我们现在就来进行详细的讨论。

RSA

RSA 是由 Rivest、Shamir 和 Adleman 在 1978 年开发的一个专利公钥加密算法。使用 RSA 有三个阶段：阶段 1 确定公钥和私钥；阶段 2 对一个消息进行加密；阶段 3 对消息进行解密。接下来将描述这三个阶段，举例说明见详细说明 11.3。



其中 K_A , K_B 和 K 是 Alice 的私钥

K_A 是 Alice 的公钥

图 11-4 公钥密码学

详细说明 11.3

RSA 三个阶段的一个例子

第一阶段:

1. 设 Alice 选择 $P=5$, $Q=11$ 。
2. 所以, $N=P*Q=55$ 。
3. $F(N)=(P-1)(Q-1)=40$ 。
4. 选择 $e=7$
(使用欧几里得算法验证 $GCD(40,7)=1$:

$$40=5*7+5$$

$$7=1*5+2$$

$$5=2*2+1$$

$$2=2*1+0$$
5. 使用扩展的欧几里得算法将 D 表示成 e 和 $f(N)$ 的线性组合。

$$1 = 5 - 2 * 2$$

$$= 5 - 2 * (7 - 1 * 5)$$

$$= -2 * 7 + 3 * 5$$

$$= -2 * 7 + 3 * (40 - 5 * 7)$$

$$= 3 * 40 - 17 * 7$$

因此 $1 = -17 * 7 \pmod{40} = 23 \pmod{40}$, $D = 23$ 。

第二阶段:

向 Alice 发送消息 $M = 25$ 。

计算 $C = M^e \pmod{N} = 25^7 \pmod{55} = 20$ 。

第三阶段:

将从 Alice 处收到的密文 $C = 20$ 解密, 密文的公钥是 $(D = 23, N = 55)$ 。

计算 $C^d \pmod{N} = M = 20^{23} \pmod{55} = 25 = M$ 。

详细说明 11.4

Diffie-Hellman 密钥交换法

Alice 和 Bob 可能使用这种方法来交换密钥, 他们必须按照如下五个步骤来实施 Diffie-Hellman 密钥交换法, 如图 11-5 所示:

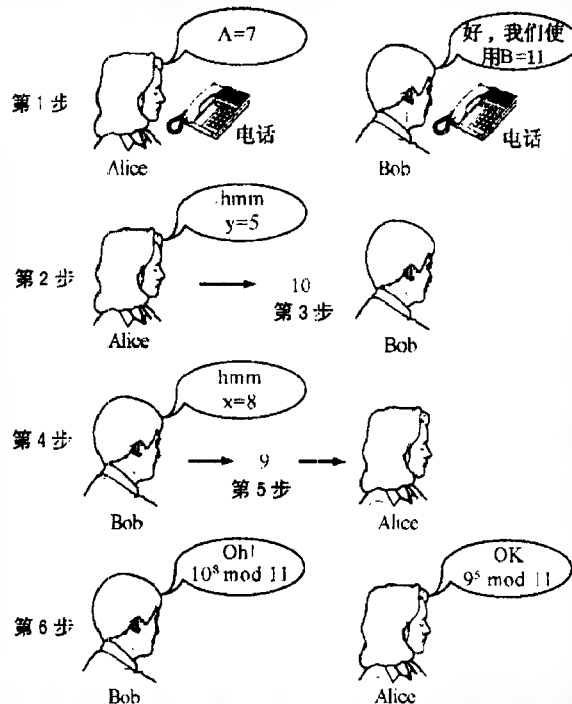


图 11-5 Diffie-Hellman 密钥交换法

1. Alice 和 Bob 同意一个公开通信的质数 p 和整数 a 。
 2. Alice 产生一个随机的数值 x : $2 \leq x \leq p-1$ 。
 3. Alice 计算 $a^x \pmod{p} = A$, 并向 Bob 发送 A 。
 4. Bob 产生一个随机的数值 y : $2 \leq y \leq p-1$ 。
 5. Bob 计算 $a^y \pmod{p} = B$ 并向 Alice 发送 B 。
 6. Bob 计算 $(A)^y \pmod{p} = (a^x)^y \pmod{p}$ 。
 7. Alice 计算 $(B)^x \pmod{p} = (a^y)^x \pmod{p} = (a^x)^y \pmod{p}$!
- 所以, $(a^x)^y \pmod{p}$ 成为了他们的共享密钥。

例如:

1. Alice 和 Bob 同意假设 $p = 11$, $a = 7$ 。
 2. Alice 产生了 $x = 5$, 其中 $2 \leq 5 \leq 11-1$ 。
 3. Alice 计算 $a^x \pmod{p}$, 或 $7^5 \pmod{11} = 10$, 并向 Bob 发送 10。
 4. Bob 产生了 $y = 8$: $2 \leq 8 \leq 11-1$ 。
 5. Bob 计算 $a^y \pmod{p} = 7^8 \pmod{11} = 9$, 并向 Alice 发送 9。
 6. Bob 计算 $(A)^y \pmod{p} = (7^5)^8 \pmod{11} = 7^{5*8} \pmod{11}$ 。
 7. Alice 计算 $(B)^x \pmod{p} = (7^8)^5 \pmod{11} = 7^{8*5} \pmod{11}$!
- 所以, $7^{5*8} \pmod{11}$ 是他们的共享密钥。

RSA 阶段 1: 决定公钥和私钥。

为了决定公钥和私钥, 每个用户必须操作以下六步。

1. 选择两个大质数 P 和 Q 。
2. 计算 $N = P * Q$ 。
3. 计算 $F(N) = (P-1)(Q-1)$ 。
4. 选择 e , 其中 $1 < e < n-1$ 并且 $\text{GCD}(e, F(N)) = 1$ (GCD 为最大公约数)。
5. 计算 d , 其中 $ed = 1 \pmod{f(n)}$ (使用扩展的欧几里得算法)。
6. 公开 d 和 n , 这些值组成了公钥。

RSA 阶段 2: 对消息加密。

为了使用 RSA 对消息 M 进行加密, 其中 $1 \leq M \leq N-1$, 必须进行如下计算。

$$C = M^e \pmod{N}$$

其中 C 是密文。

发送 C 。

RSA 阶段 3: 对密文解密。

为了使用 RSA 对密文 C 进行解密, 必须进行如下计算。

$$C^d \pmod{N} = M$$

其中 M 是原来的明文。

使用公钥加密的数字签名

数字签名的公钥加密方法使用 RSA。在此方法中, 创建者使用它的私钥来加密它的整个数据文件 (代价高昂) 或者它使用摘要函数所产生的文档签名。相对于私钥加密方法, 使用

公钥加密的主要优点是它不存在密钥分发的问题。这个方法假定可以相信发布公钥的来源 (参见 11.3.1 节), 然后接收者可以使用公钥来解密签名或文档来验证它的来源和/或内容。由于公钥密码学的复杂程度 (参见图 11-4), 只有正确的公钥才能解密消息或摘要。最后, 如果你发送消息给已经知道公钥的某人, 那么就可以使用接收者的公钥将消息或摘要加密, 这样只有接收者才能使用他们的私钥来验证信息的内容。

11.2 身份认证

在一个分布式环境中提供身份认证机制有几个步骤是必须的。第一步是认证或检验用户的身份。完成这一步有三个基本的方法 [Sha77, WL92]。首先, 可以通过用户所知的一些信息如口令来检验用户的身份, 这是最普通的方法, 但不是最安全的。第二个方法使用用户所持有的一些信息如密钥等。第三个方法包含了用户自身的一些信息, 比如用户的指纹或是视网膜模式等, 这是最安全、最昂贵的方法了。注意: 到现在为止, 这些方法只是在集中式系统中能很好地工作。一个分布式的操作系统一定要解决好以下问题。

1. 窃听 (eavesdropping): 我们怎么来防止有人从通信线路窃听信息?

2. 多口令管理 (multiple password management): 如果我们访问一个多机系统, 是不是每个系统都要保持用户 ID 和口令的一个副本呢? 每个存储认证信息的数据库都有可能泄漏, 从而导致系统安全的崩溃。此外, 是不是每次我们要完成一项工作的时候都必须出示口令呢?

3. 重演 (replay): 认证信息在网上传播的时候, 即便它是加密的, 别人也可以复制它, 然后过一段时间后重新发送, 从而导致不适当的访问。

4. 信赖 (trust): 认证行为是单向的还是用户也有能力去验证并信任将会合法地执行服务。一个集中式系统可以信赖其自身, 而分布式系统则一定要找到一个方法去相信其他人。

通常这些问题的解决办法是使用一个认证证书管理系统。认证证书是分布式系统中由计算机产生的, 通常具有时间有效期的, 经过认证同意后可以方便地访问多种资源的信息包。时间有效期的特性预防了一段时间后重演的发生。可以经由一个时间戳或者一个临时标签 (nonce) 来实现。临时标签是一个每次通信中都独一无二的一个随机值, 所以没有两次通信使用相同的临时标签, 这样就非常容易地将重演检测出来。认证证书管理有两个基本的方法。11.2.1 节中所示的第一个方法利用认证证书列表, 此表包含了从可信的认证权威处得到的认证信息的一个列表, X.509 就是一个例子。任何需要检验身份的服务必须检查列表以检验客户的可靠性。11.2.2 节中所示的第二个方法包含了一个集中式的证书分派中心, 客户在此处获得所需服务的各种认证信息。以后客户使用服务的时候就显示这个服务的认证信息。

11.2.1 证书表

证书表基于如 11.1.2 节所示的公钥加密法。用户的身份标识信息成为一个认证证书, 可以包括在证书表中。认证部门检验用户的身份就是指认证授权和检验用户的公钥。图 11-6 描述了一个服务是如何利用证书表的, 例如 RPC。

认证证书是通过认证部门的数字签名来认证的, 数字签名参见 11.1 节, 可以有若干个认证部门。我们现在以 X.509 为例, 研究一下证书表是如何发挥作用的。

X.509

X.509 是第 7 章中讨论的 X.500 目录服务的身份认证部分。目录服务提供了证书表的位

266
270

271

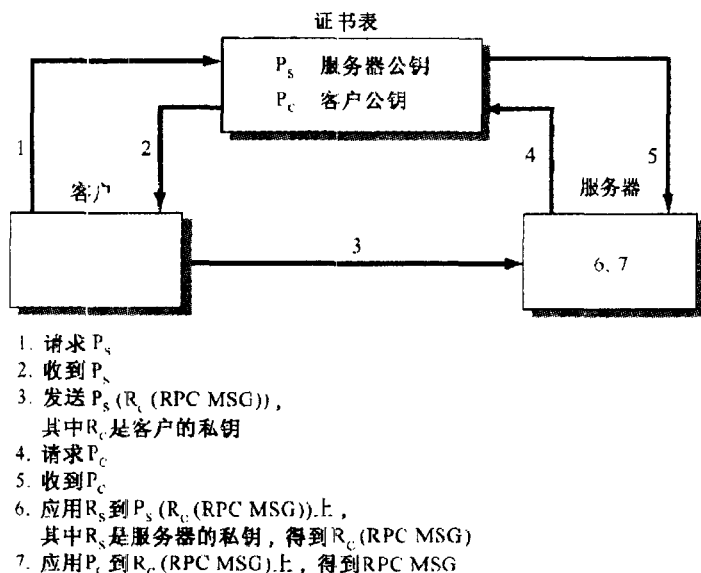


图 11-6 用做安全 RPC 的证书表

置,但是它假定存在一个可信赖的认证部门来创建这些认证证书。认证证书是由发布人签署的(以发布人的私钥加密),由此将证书持有者的姓名与发布人的公钥联系在一起。以下是 X.509 认证证书(版本 1)中所包含的元素。

V: 版本号。版本号将 X.509 的几个后继版本区分开来,默认值是 1988 年的版本。

SN: 序列号。序列号是发行认证证书的部门发布的一个独一无二的整数值,序列号是和认证证书明确联系的。就像社会保险号与美国公民或移民是明确联系的一样。

AI: 算法标识符。算法标识符标识了认证部门签署认证证书时所采用的算法,认证部门用它的私钥来签署每一份证书。

CA: 发行者或认证部门。就是由认证部门来创建认证证书的。

$TS_{\{A\}}S$: 有效期。提供了有效的最初和最迟日期,就像信用卡上的日期一样。

A: 正文。证书所要证明的标识。

Ap: 公钥信息。为证书所标识的正文提供公钥和算法标识符。

签名: 认证证书的签名包含了认证的其他各个部分。签名是其他数值域的一个 hash 函数值,并使用认证部门的私钥进行加密以保证整个认证证书信息的完整性。如果有人使用认证部门的公钥将 hash 值解密,并对整个证书计算出来的 hash 值进行比较,如果两者不相符合的话,则说明这个证书的内容被非法更改了。

任何拥有认证部门公钥的用户都可恢复并验证证书表内每一个认证证书的真实性。X.509 提供了三个使用证书表的身份认证过程: 单路认证,二路认证和三路认证。

单路认证

单路认证保护消息的整体性与原始性。可以通过用户使用自己的签名(私钥)签署的时间戳、临时标签值和目标位置标识等信息来实现。此时接收者可以将信息用经由证书表验证的创建者的公钥进行“反签名”,从而检验信息的内容。因为仅限于在目标位置认证,所以就称为单路认证。

二路认证

二路认证允许接收者或者说目标也可以由发送者或创建者进行验证。除了单路认证的过程以外，目标也向创建者发送一个答复。答复包含了新的时间戳、原先的临时标签以及一个新的临时标签。答复使用创建者的公钥来签署。因为公钥加密法中只有对应的私钥，所以说只有创建者的私钥才能将答复解密。此外，临时标签一定要是原先的临时标签，否则将不能确认。

三路认证

当目标和创建者不具备同步时钟或者不愿相信时钟时就采用三路认证。除了二路认证的过程之外，创建者然后发送一个目标答复的答复，其中包含原答复中的新的临时标签，如图 11-7 所示。一旦验证了临时标签值是相符合的，就没有必要验证时间戳了。

证书废除

证书表的使用过程中，存在一个潜在的问题是认证证书过期之前的证书废除。这通过保持一个废除列表来得以实现。这个列表是与证书表是一同保持的，当需要确认证书是否真实有效时可以查询该表。这与过期信用卡手册相类似，后者保持了一个在有效期前失效的信用卡的列表。废除表

还保持一个由证书部门的私钥签署的消息摘要以保持并确保列表的完整性。

X.509 中使用多个证书认证部门

另一个潜在的问题就是多个证书认证部门的存在。某人使用了一个不同的证书认证部门，而这个部门已列在了你的部门列表中，这样你就可以验证他的身份。为此，必须用你的部门来验证其他的部门的身份。一旦完成，你就拥有了其他部门的公钥，而你也就验证在部门列表中的用户的身份，如图 11-8 所示。这称为连锁证书授权，可以与数学上的传递

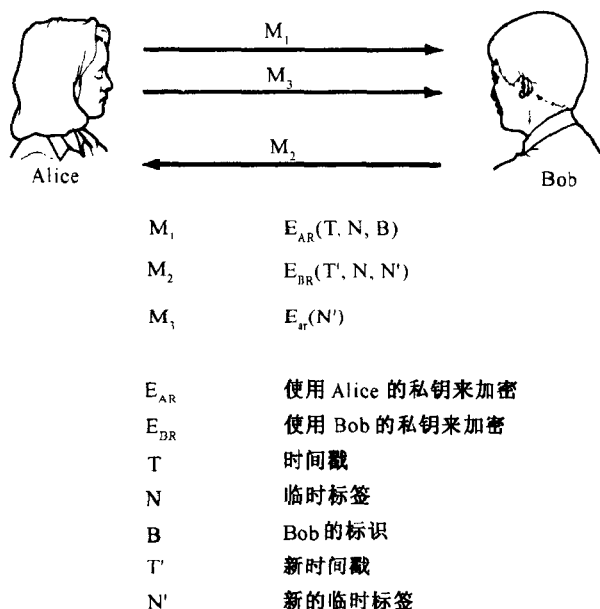
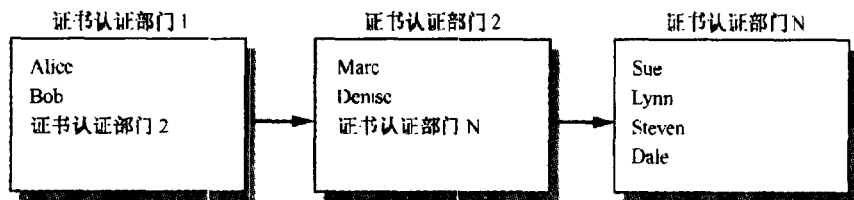


图 11-7 三路 X.509 认证



Alice 是怎样验证 Steven 的公钥的

1. Alice 知道证书认证部门 1
2. 证书认证部门 1 知道证书认证部门 2
3. 证书认证部门 2 知道证书认证部门 N
4. 证书认证部门 N 知道 Steven
5. 结论: Alice 可以信赖并知道 Steven 的公钥

图 11-8 X.509 中的连锁证书认证部门

定律来进行比较。如果 Alice 知道证书认证部门 1，证书认证部门 1 知道证书认证部门 2，而证书认证部门 2 知道证书认证部门 N，后者知道 Steven，所以 Alice 知道 Steven。

认证的种类型

最后，X.509 提供了不同种类的认证。每一种都通过了不同程度的验证，所以反映了证书表中授权信息不同等级的可信度。

最低的等级是一类认证。在这一类认证中，使用者提供电子邮件地址，而认证部门向这个地址发送回应，以这种方式来检验身份。最高的等级是四类认证，它要求使用者以物理形式出现，认证部门必须进行一次背景的核查。X.509 证书可以被安全套接字层（SSL）这样的协议使用，详细说明 11.5 进行了相关描述。

详细说明 11.5

安全套接字层（SSL）

SSL 起始于 Netscape，最初的目的是为了在 Web 页上安全地通信，随后它被大多数浏览器所支持。最近，WinSock 也包含了 SSL。互联网工程任务组（Internet Engineering Task Force, IETF）正式开发了 SSL 标准。它的目标是确保私密性、整体性和真实性。它的应用从原本的浏览器领域，发展到开发公司合作工程内的安全传输和安全通信等。SSL 的使用允许两个实体以某种（依赖于密钥的长度）安全方式在一个公众网络上通信，并且不需要很多的已有代码的重写工作。这个方法很可能比寻求并利用一条专线连接的方法效率更高，效果更好。

SSL 包含三个阶段。第一阶段是认证握手（authentication handshake），而且要完成以下的任务。

1. 建立一个双方用来通信的协议。
2. 定义一个加密算法。
3. 决定是否要采用数据压缩。
4. 交换密钥。
5. 可以使用 X.509 认证相互的标识，包括使用 X.509 中的连锁证书。

实际发生的数据通信是在第二阶段。数据以加密的形式发送，如同第一阶段所制定的。另外，每一个数据段都要经过数字签名，通常是使用 DES。最后一个阶段简单包含了执行一个附加的握手以确保双方均能意识到连接结束了。最后，关闭连接。一般来说，SSL 使用一个 40 位的密钥，因为现在美国的贸易禁令禁止更长的密钥用做出口。

11.2.2 集中式证书分送中心

一个集中式的证书分送中心依赖某一个地点来管理所分送的全部认证证书。所以，证书分送中心就成为分布式系统中一个关键的因素。如果证书分送中心崩溃或受到危害，整个分布式系统都要受到影响。这个地点在认证数据库管理器中保存了一份所有合法用户的私钥和系统服务的拷贝。证书分送中心使用这些私钥向每个想利用服务的用户分发这些系统和服务证书，这些服务证书必须以客户使用服务的形式表现出来。对每一个用户和每一个服务，都

有一个独一无二的服务证书。这些服务证书与集中式系统的能力安全概念是类似的，所以，一个用户可能有若干个服务证书，而每个服务证书都必须对应使用一项服务。服务证书只在一个有限的时间内对特定的服务有效。如果用户的服务证书过期了，用户必须使用用户的系统认证证书从证书分送中心处获得一个新的服务证书。我们现在通过 Kerberos v.5 来研究一个集中式的证书分送中心是怎样工作的。

Kerberos v.5

Kerberos 是第一个广泛应用的分布式认证协议，并对此后的协议产生一定的影响，如 SESAME 和 KryptoKnight，现在可以从 <ftp://athena-dis.mit.edu/pub/kerberos> 免费得到，在 [NeTs94] 中有所描述。它的目标是提供一个可信的第三方认证系统，通过使用第三方，分布式系统中的每个位置都减轻了负担，否则它们都需要保持一个复杂的用户标识和口令的数据库。Kerberos 取而代之地只保持了一个认证信息的主数据库，称为 Kerberos 数据库管理系统 (KDBM)。可能有几个 Kerberos 的密钥分送服务来辅助 KDBM，它们包含了主数据库的只读拷贝以帮助 KDBM 来缓解瓶颈的问题。Kerberos 密钥分送服务也允许系统在 KDBM 暂时不可达的情况下仍然发挥作用。Kerberos 系统包含了以下的四个阶段，将在详细说明 11.6 中作进一步讲解。

阶段 0: 注册 Kerberos

在任何一个用户建立会话之前，首先必须离线 (off-line) 地在 Kerberos 密钥分送中心建立用户的标识。完成之后，用户的 ID 和口令在 Kerberos 数据库管理器中以加密的方式保存着。这样就可认为用户已经注册了，并可以准备利用 Kerberos 协议的网络服务。

275

276

详细说明 11.6

Kerberos 身份认证服务 v.5

令:

ID_T = 票据授予服务的 ID

ID_C = 客户的 ID

ID_S = 服务器的 ID

N_i = 临时标签值

K_C = 客户的私钥

K_S = 应用程序服务器的私钥

K_T = 票据授予服务器的私钥

K_1 = 客户与 TGS 共享的密钥

K_2 = 客户与服务器共享的密钥

T_1 = 系统票据

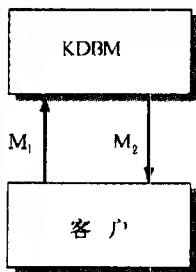
T_2 = 服务票据

T_s = 开始时间戳

T_e = 结束时间戳

$E(a, K)$ = 对 a 使用密钥 K 进行加密。

阶段 1 如图 11-9 所示，阶段 2 如图 11-10 所示，阶段 3 如图 11-11 所示。



其中:

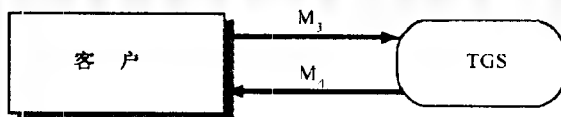
T = 票据

$M_1 = (ID_C, N_1)$

$M_2 = E((N_1, K_1, C_1), K_C)$

$C_1 = E((ID_C, ID_T, T_{s1}, T_{e1}, K_1), K_1)$

图 11-9 Kerberos 阶段 1 详解



其中:

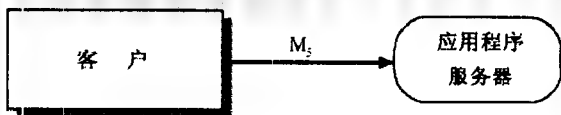
$M_1 = (ID_s, N_2, C_1, C_2)$

$C_1 = E((ID_C, T_1), K_1)$

$M_2 = E((N_2, K_2, C_2), K_1)$

$C_2 = E((ID_C, ID_s, T_{s2}, T_{e2}, K_2), K_s)$

图 11-10 Kerberos 阶段 2 详解



其中:

$M_3 = C_4, E((ID_C, T_3), K_2)$

图 11-11 Kerberos 阶段 3 详解

阶段 1: 获得一个系统票据

Kerberos 认证证书称为票据。第一阶段可以获得一个系统票据，它用来从第二阶段中的票据授予服务 (Ticket Granting Service, TGS) 处获得服务票据。为了获得系统证书，必须在 Kerberos 中注册 (阶段 0)。获得的 Kerberos 系统票据以 TGS 的特别私钥加密。这个系统票据中有客户的身份信息，如下所示：

- ◆ TGS 和客户间使用的暂时会话密钥。
- ◆ 客户标识。
- ◆ TGS 的标识。
- ◆ 票据过期时间。
- ◆ 客户的网络地址。

票据经过 Kerberos 加密, 所以只有 TGS 有权验证系统票据是合法的。任何非法的票据都不可能使用 TGS 的私钥加密, 只有 Kerberos 和 TGS 才知道密钥。系统票据与 TGS 的标识一起发送给客户, 一起发送的还有 Kerberos 产生的给 TGS 和客户使用的会话密钥, 以及发回的客户发送给 Kerberos 的临时标签值。如果临时标签值与发送给 Kerberos 的值相符合, 客户就知道系统票据是合法的, 并且是最新的。

阶段 2: 获得服务票据

为了获得服务票据, 客户向 TGS 发送一个经过会话密钥加密的数据包, 会话密钥是由客户与 TGS 共享的。这个数据包内包括客户名称、网络地址、临时标签值以及客户想使用的服务的名称。另外, 发送的系统票据是非加密的。

当 TGS 使用密钥将系统票据解密后, 它可以验证系统证书中的信息是否与接收到的数据包中的信息相符合。如果符合, TGS 就可以确定请求是合法的。认定为合法请求之后, TGS 准备向客户发送一个数据包, 数据包内包括了使用服务器的私钥加密的服务票据 (这样, 只有服务器才可以验证此票据)、服务的名称、有效时间以及一个临时标签。这个数据包是由客户与服务器共享的会话密钥加密的。如果临时标签值相符合, 客户就知道这是正确的回应。我们现在已经有了使用 Kerberos 中所列的某项服务的密钥。

阶段 3: 使用间接服务

现在客户有了服务票据, 准备使用服务。服务票据允许此项服务对客户的标识进行身份认证。为了使用服务, 客户向服务方发送一个数据包, 包中包含了服务票据和客户的标识。我们知道服务票据是以服务器的私钥来进行加密的, 并且包含了客户的身份标识和时间戳。如果时间戳没有过期并且身份标识也符合, 那么客户就通过了身份验证, 可以自由地使用服务了。

11.3 访问控制 (防火墙)

当一个开放的分布式系统设计成允许信息在所有相连接系统间自由流动时, 分布式操作系统必须提供访问控制的功能, 这样只有需要共享的信息才能共享。此外, 一个系统所需要的, 其他的系统未必需要。一个公司也许会采用这样的安全策略: “只要没有明确标明是允许的, 那就不允许”。同时, 一个大学或者个人网站则可能恰恰相反, 安全策略是 “只要没有明令禁止, 那就是允许的”。虽然访问控制是一个操作系统的概念, 但当今的分布式操作系统必须依赖硬件的协助, 通常采用所谓的防火墙技术来实现访问控制。

防火墙必须避免系统受到安全方面的威胁, 并且防止所有的安全方面的威胁通过此墙而进入到它所保护的系统内。同时, 防火墙决不能影响任何遵循组织中的安全策略的行为。

防火墙一般归入以下两个种类:

1. 包过滤网关。
2. 代理服务。

一个地点的系统可能经常同时采用两种防火墙以实现所要求的访问控制。我们分别来研究一下防火墙的两种基本种类, 然后了解一些合并两种方法的基本防火墙体系结构。

11.3.1 包过滤网关

一个包过滤网关防火墙包括了一个安全工程师, 他必须明确指出什么可以通过防火墙, 包括有哪些内部信息是可以流出防火墙的, 以及允许哪些外部站点通过防火墙。另外, 安全

工程师可以配置防火墙，以指定哪些内部的计算机服务可以与外部世界共享。

包过滤网关防火墙一般在连接内部系统与外部世界的路由器上实现。一般的路由器都可以实现包过滤的功能，但是防火墙路由器趋向于提供更友好的用户界面，可以更方便地对基于安全的过滤进行配置。就像遍布世界的邮政服务需要信封上的地址一样，网络也需要信息带有自己的地址。防火墙检查的就是这些地址，在信息从某一方向通过之前把它们与自己的列表相比较。因为所有的信息都必须通过执行防火墙功能的路由器，所以所有的信息都要被防火墙所检查。

包过滤网关的规则可能如表 11-3 所示。

表 11-3 包过滤网关规则

动作	目的地址	目的接口	源地址	源接口
禁止	us.net		Enemy.net	
同意	us.net		Friend.net	

这些规则禁止了所有从 enemy.net 到 us.net 的信息，无论连接的是网络的哪一个接口。此外，这些规则允许所有从 Friend.net 到 us.net 任何接口的信息。具体的格式因为特定的路由器而有所不同，并且依赖于它的制造商。无论是哪个制造商，设置系统的安全工程师必须明确指出允许的和禁止的所有事项，这并不是容易确定的。有些服务，像 UNIX X11 窗口管理器，对内部用户是可用的，然而如果有内部的用户可以使用 X11，就不可能禁止外部的用户使用了。一个未经授权的用户可以执行内部用户的屏幕转储与记录击键信息，这是一个非常严重的安全隐患。另外，防火墙上留有的漏洞可能会使系统的安全保护工作毁于一旦。

280

11.3.2 代理服务

代理服务是对内部客户提供的访问外部世界的服务。提供这项服务的同时，它还增加了一些安全的措施。代理服务有两种基本的类型：

1. 应用层次的网关代理服务
2. 电路层次的网关代理服务

应用层次的网关防火墙通过重写所有主要的应用程序来提供访问控制。新的应用程序位于一个所有用户都可以使用的集中式主机上，这些主机以中世纪高度坚固的城堡来命名，称为**设防主机**（bastion hosts），代表关键的安全点。这些主机经常是**双重位置**的主机（dual-homed）或者是位于多个网络的主机。应用程序运行的时候和它们修改以前没什么不同，只是消除了安全上的隐患，特别是，新的应用程序增加了一个非常重要的小特性：通过前面讨论过的方法之一来进行身份认证。应用网关防火墙是对包过滤网关出色的补充，并且可以用来重写像 X11 这样的应用程序。

电路层次的网关与应用层次的网关有相似之处，两者都是为单个应用程序设计的。不同的是它们对用户是透明的，特别是外部的用户可以通过 TCP 端口与网络连接。在电路层次的网关中，防火墙提供了 TCP 的端口并来回传递数据字节，就好像一段线路一样。通过这样的方式就完成了回路，同时也不影响应用程序的协议。

既然它们是在一个较低的层次上运行的，电路层次的网关需要修改客户以获得目的地

址，而这在应用层次的网关中是随时可用的。修改后的客户经常只用做外部的连接，所有的过滤动作都是单独对源地址和目的地执行的，并没有特定命令的附加信息。

除了传递数据字节，修改后的客户和电路层次的网关保存了一份所传递字节数量的日志，日志中还有 TCP 的目的地。如果一个已知的站点有安全上的问题，系统管理员可以使用这份日志去通知系统中的任何不幸与这个站点连接的成员。

281

11.3.3 防火墙体系结构

我们现在来研究三个融合了包过滤防火墙和代理服务防火墙的基本防火墙体系结构。

设防主机体系结构

最简单的体系结构是设防主机体系结构，这种设计专门使用了一台设防主机以提供代理服务。尽管主机有能力为网络间信息的传递提供路由，但并不推荐使用主机的这个特性，因为可能因滥用它而经常导致淹没防火墙。所有的本地系统被认为是内部系统，所有的非本地系统被认为是外部系统。设防主机是一个双重位置的主机，处于内部系统和外部系统之间。它不直接属于任何一个网络，而是作为网络间的网关来运作。所有的内部系统可以与设防主机通信，所有的外部系统也可以与同一台设防主机通信。内外系统不能直接相互通信，而是将主机作为各自的代理服务器来实现通信，如图 11-12 所示。如果主机从外部的连接处收到一个包，包上的地址是内部地址，那么这个包肯定具有欺诈性。这个体系结构展现了代理服务所有的弱点，包括它们所能提供的服务的局限性。

282

过滤主机体系结构

过滤主机体系结构使用了一台有代理服务的设防主机，还有一个路由器用来进行筛选并提供包过滤能力。与先前的体系结构不同的是，设防主机处在内部网络中，如图 11-13 所示。

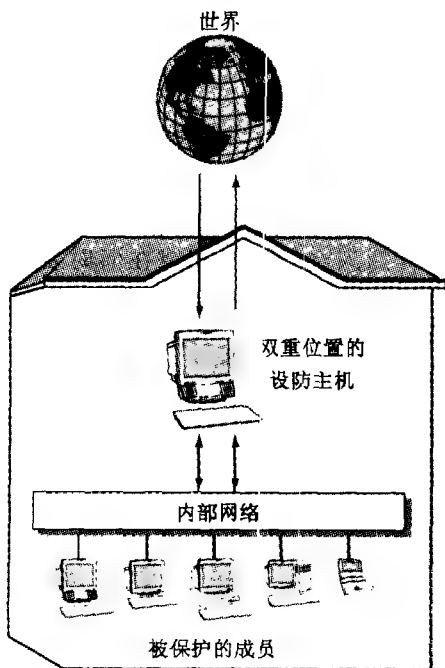


图 11-12 设防主机防火墙体系结构

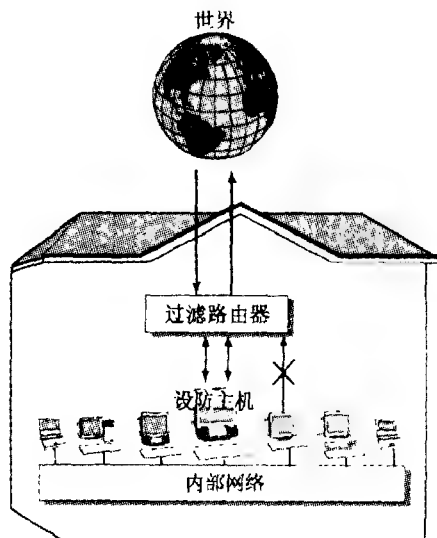


图 11-13 过滤主机防火墙体系结构

过滤路由器上的包过滤能力将所有允许的有内部目的地的外部通信都引向了设防主机。同时，包过滤可以设置成只允许设防主机可以访问外部。如果设防主机以外的内部主机都被禁止访问包过滤路由器，内部的主机将被迫使用设防主机的代理服务来访问外部的主机。这样，一个有严格的安全要求的网站将会禁止内部的主机直接通过包过滤防火墙访问外部的网络。

过滤子网体系结构

三种防火墙体系结构中最安全的是过滤了网体系结构。这种体系结构包括两个包过滤防火墙和一个代理服务防火墙。代理服务防火墙或者设防主机是和它自己的网络相连的，它自己的网络位于两个包过滤防火墙之间。一个包过滤防火墙将设防主机的子网与外部网络相连接，另一个包过滤防火墙将设防主机的子网与内部网络相连接，如图 11-14 所示。

能访问外部网络的子网称为过滤子网。因为

对于大多数本地区域的网络，所有网络上的主机都可能“偷窥”或者看到所有网络上的信息交流，所以这种体系结构的最大好处是在过滤子网上减少了只在内部流通的信息。这样，一个入侵者必须通过外部包过滤路由器，破坏设防主机，再通过内部包过滤路由器才能破坏访问控制。

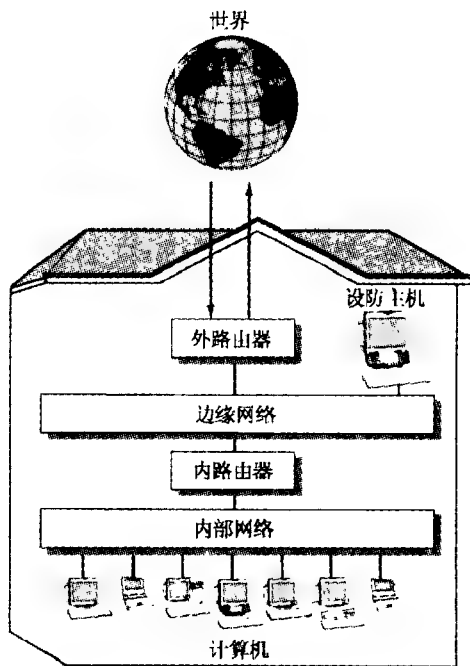


图 11-14 过滤子网防火墙体系结构

283

11.4 小结

这一章讨论了已有的、安全的独立系统在向网络连接时增加安全措施的方法。与集中式的系统一样，分布式的安全包括了身份认证和访问控制，另外还需要考虑加密和数字签名的使用。当决定使用什么方法的时候，必须考虑把要保护的价值的价值放在首位，还必须考虑信息的时间范围。例如，Microsoft™ 多半不会再担心什么人会知道他们将在何时发布 Windows95™，因为它已经发布了。然而，当这个产品正在首次酝酿、计划、开发的过程中，这个信息无疑是非常重要的，必须非常秘密地保护起来。因此，网络上一个给定的系统可能只使用包过滤网关，而其他系统则同时选择了使用代理服务器。同样，不是所有的系统都如此复杂和昂贵以致要使用四类证书授权。然而像 Equifax 这样的公司是用来维护美国公民的信用卡数据库的，它就选择了使用一个隔离的网络，它认为即使使用了当前所有的分布式安全技术，任何与外界网络的连接仍然都是非常冒险的。总之，就像在独立的计算机上使用的标准操作系统一样，有可能达到更高层次的安全性，但仍不具有 100% 的保证。

284

11.5 参考文献

以下的参考文献可以提供有关分布式安全的更多的信息：[Bra95, Car95, ChBe94, ChZw95, KPS95, McC98, Pf97, Pow95, Sta95 以及 Wil94]。一些相当经典的有关这一章信息

的研究论文包括 [ABCLMN98, AbPo86, ADFS98, AGS83, AJP95, And94, BAN90, BBF83, BFMV84, ClHo94, DH76, DESA81, DoMc98, Erick93a, Hel78, Klu94, LABW94, Lai92, Nes83, NeTs94, NIST77, NIST93B, NIST94a, NIST94b, Opp97, Opp98, PaSh98, RSA78, RuGe98, Schn98, Sim92, Sha77, THPSW98 和 WoLa92]。

以下提供了有关安全问题的一些 Internet 上的重要资源的起始链接。一个有关安全性的列表链接可以在 <http://www.semper.org/sirene/outsideworld/security.html> 找到。这个网站包括了标准和协议的链接, 以及很多包含其他资源的网站。通过向 rsaref@rsa.com 发送邮件, 可以获得有关如何得到 RSAREF 的信息。RSAREF 是一个免费 (只对美国和加拿大公民) 的教育工具, 用 C 写成, 可以执行 RSA 的加密和密钥生成、DES、Diffie-Hellman 密钥协定、三重 DES 等一系列特性。Kerberos 所需的备注 1510 文档可以在 <ftp://ftp.isi.edu/innotes/rfc1510.txt> 找到。有关怎样 (免费地) 得到 Kerberos 的信息可以在 <http://web.mit.edu/kerberos/www/krb5-1.0/announce.html> 找到。一个有关 Kerberos 的论文和文档的列表可以在 <http://nii.isi.edu/info/kerberos/documentation.html> 找到。可以在 <http://www.esat.kuleuven.ac.be/~vdwauver/sesame.html> 找到有关获得 Sesame 的信息及其源程序的信息, 这个程序是一个建立在 Kerberos 上并扩展了 Kerberos 的分布式访问控制的程序。一个带有 SSL 的参考资料指针的 SSL 免费版本则位于 <http://psych.psy.uq.oz.au/~ftp/Crypto/>。在 <http://www.ssh.fi/tech/crypto/> 有一系列与密码学有关的算法和标准的链接。

习题

- 11.1 对分布式系统来讲安全性还要考虑哪些方面? 有可能提供与独立的集中式系统同等程度的安全吗?
- 11.2 讨论对称加密与非对称密钥加密的优缺点。
- 11.3 使用 11.1 节中列出的 RSA 公钥加密算法, 令 $P=7$, $Q=11$, $e=4$ 。
 - a. 计算 D 的值。
 - b. 公钥是怎样的?
 - c. 如果消息是 $M=17$, 发送的加密后的信息是怎样的?
 - d. 使用公钥解密以验证消息。
 - e. 计算详细说明 11.4 中的共享密钥。
- 11.4 写一个程序来实现详细说明 11.4 中的 Diffie-Hellman 密钥交换算法。
 - 选项 1: 允许 Alice 和 Bob 使用同样的终端来交换密钥, 允许程序选择随机数。
 - 选项 2: 假定用户是 Alice, 程序是 Bob, 允许 Alice 和 Bob 交换数值信息。
 - 选项 3: 使用第 4 章中学到的 IPC 机制, 允许两个在不同计算机上的用户交换密钥和数值信息。
 - 选项 4: 允许用户选择以上三个选项中的任何一个。
- 11.5 讨论 X.509 中单路、两路、三路身份认证的相对优点。
- 11.6 讨论以下 X.509 身份认证过程的可能用途。
 - a. 单路身份认证
 - b. 二路身份认证
 - c. 三路身份认证

- 11.7 描述一个 SSL 第一阶段中消息交换的可用算法，使用四个密钥并允许对对方的身份标识进行验证（同一给定用户的公钥和私钥分别作为两个密钥）。
- 11.8 讨论分布式身份认证中使用证书表和证书分送中心的相对优缺点。
- 11.9 Kerberos 中有若干个密钥分送服务有什么优点？
- 11.10 在集中式系统中，经常使用一个访问矩阵，矩阵的行表示用户（或用户集），矩阵的列表示资源（拥有不同程度的颗粒度定义）。矩阵中的信息定义了用户是否有资格使用此项资源，或者在多大程度上有资格使用。对以下每个选项，描述一下如果这种矩阵用在分布式系统中会产生什么问题。
- a. 在一个单独的集中式服务器上实现。
- b. 在有多组这样的矩阵拷贝的分布式系统中实现（提示：考虑第 7 章的相关内容）。
- 11.11 一些节点喜欢使用多个设防主机。讨论这种方法的优点。
- 11.12 本章中所述的三种防火墙体系结构中，最安全的是哪一个？为什么？
- 11.13 一般都认为在筛选后的子网防火墙体系结构中使用多个外部路由器是没有问题的，但是使用多个内部路由器则被认为冒着极大的安全风险。为什么？
- 11.14 使用电缆调制解调器提供家庭访问的体系结构称为电缆调制解调器体系结构，其中单条电缆在每个家庭内进出（非常像珍珠项链），同邻居一起连成了一个虚拟的局域网。
- a. 这种实现方式将出现安全上的什么基本问题？为什么？
- b. 举一个实例说明家庭中使用电缆调制解调器访问 Internet 将会如何破坏团体的安全。
- 11.15 口令卡是一种像信用卡一样的口令记号产生器，它也有其他很多的名字为人所知。确切的记号是依赖于当前时间与卡的标识的，产生的记号与适当用户的标识一同组成了某种授权。当进入一个系统或者一幢大楼时，这样的卡可以用做身份认证。基于学习过的有关同步的知识，以及在这一章中学到的知识，指出随着这种口令卡安全系统的实现而一同产生的两个优点和两个缺点（或者不足）。为什么在数据在网络上传输并不安全的环境中这种技术非常关键？
- 11.16 分布式安全经常要面对效率与易用性之间的平衡。讨论四个能够显示这种分歧的例子。

第 12 章 实例研究：Windows 2000

Windows 2000 是基于 Windows NT 技术的，在早期开发阶段参照了 Windows NT 5.0。对于这个流行的 Windows 产品，经常提及的一个重要特点是支持分布式计算，所以人们喜欢并广泛选用它作为分布式操作系统的一个实例。1989 年微软 NT 的子系统设计基本理论声称 Windows NT 将是一个可移植的操作系统，具有灵活性，健壮性，安全性（全时的）和易维护性。Windows 2000 的一个贡献是它结合并采用了一系列的标准，从而能与其他系统进行相互操作，从而提高了作为分布式操作系统的能力。本实例研究将给出其中的一些特性，主要包括以下内容：

- ◆ 微内核设计。
- ◆ 硬件抽象层。
- ◆ 即插即用。
- ◆ NT 文件系统。
- ◆ 活动目录以及 LDAP 和 DNS 的使用。
- ◆ 修改日志。
- ◆ 索引服务器及其与 HTTP 的兼容性。
- ◆ 微软管理控制台。
- ◆ 集群服务器。
- ◆ 安全配置编辑器。
- ◆ 加密文件系统。
- ◆ 微软安全支持提供者接口及其与 DES 和 RSA 协同工作的能力。
- ◆ 与 DCOM 的结合。

289

即使有许多不同的 Windows 2000 组件，仍然很少有人知道实际上有四种 Windows 2000 操作系统。

1. Windows 2000 专业版，用来替代 NT 工作站。
2. Windows 2000 服务器版，替代 NT 服务器功能，并提供了少量的超集。该版本提供了诸如 Web 服务器的应用等。
3. Windows 2000 高级服务器版，它继续提供对网络和分布式计算的支持，包括对集群技术和负载平衡的支持。
4. Windows 2000 数据中心服务器版，是最大的超集，最多能支持 64GB 的物理内存。

在本实例研究中，讨论了 Windows 2000 技术的整个超集——Windows 2000 高级服务器和数据中心服务器。实例研究的材料取自微软 Windows NT 白皮书系列，包括 Windows 2000 和有关技术 [Micr98]。在本书交付印刷时，Windows 2000 完整的产品版本尚未能完全提供。Beta 第 2 版及随后推出的产品结合了一些重要的改变，包括即插即用的并入、活动目录、微软管理控制台以及分布式安全扩展等。为了与过去的内容相衔接，本实例的某些内容仍参照早期的版本。12.1 节叙述了 Windows 2000 基本技术的设计目标。12.2 节给出了 Windows 2000

微内核的概述，包括 Windows 2000 地址空间的信息。12.3 节说明了 Windows 2000 中即插即用的实现。Windows 2000 文件系统在 12.4 节中讨论。而活动目录则在 12.5 节中叙述。微软管理控制台是 12.6 节的主题。Windows 2000 高级服务器版中提供的集群技术在 12.7 节中描述。12.8 节集中说明 Windows 2000 基于安全的特性。12.9 节给出了瘦客户的简短描述，它的内部代码名称为 Hydra。

12.1 概述：Windows 2000 设计

微软 Windows 2000 操作系统有五个基本的设计目标，继承了 Windows NT 的基本技术。本节中将枚举这些目标并讨论在设计中如何实现它们。第一个目标是健壮性。大多数人都同意如下意见：相对于其他的 Windows 操作系统，Windows NT 家族的操作系统更为健壮。关于一个健壮系统的特征包括一个系统抵御操作系统内部故障，以及偶发的和故意的外部故障的能力。这一点对于那些准备使用分布式的系统、与网络连接的系统、可能遭受外部事件干扰的系统是有益的。一个系统要成为健壮的，必须以一种可预测的方式运作，并且要基于良好定义的界面和系统行为。微软提出的下述五个决策能帮助实现健壮性目标。

1. 内核模式能提供定义良好的应用程序设计接口 (APIs)，并力图使编程所需的参数和图形标记最少。该决策可提高应用程序实现、测试和文档编制的方便性。

2. 整个系统采用规范化设计，设计的文档在编码前完成编制。

3. 主要的组件，如 Windows 32、OS/2 和 POSIX 被分割成独立的子系统，这种简洁而优美的设计风格可使每个子系统只需专心实现 API 集所要求的功能。

4. Windows NT 及其子系统采用基于帧的例外处理程序，从而能采用可靠而有效的方法来指明程序设计中的错误和坏的以及不可访问的参数。

5. 可能最受关注的决策是将操作系统分成核心模式系统服务和子系统。该决策的结果能避免不良行为的应用引起操作系统的崩溃。

基于 Windows NT 技术的第二个设计目标涉及到可扩充性和可维护性。它期望系统的设计能适应计算机、计算机相关制造商（包括微软自身）、原始设备制造商 (OEM) 的应用以及未来可预见的用户的需求。就此观点出发，可能会与将 Windows 2000 设计成支持分布式计算的目标不一致。下面五个决策帮助实现第二个目标。

1. 系统的设计是简单的和文档化的，系统开发遵循公共编码标准，这样就不需要依靠“民间传说”来维护系统了。

2. 系统的每个主要部分作为子系统实现，由此提高了 Windows 2000 的隔离与控制独立的能力。例如，如对 POSIX 进行改动只需改变 POSIX 子系统。可以增加一个子系统来提高系统的可移植性，且允许在一个分布式环境中，增加一个组件类型而不会影响已存在的不使用子系统的早期版本。

3. 子系统设计允许附加子系统。增加的子系统并不需要对原始系统或操作做任何改变，该类设计应限制或至少大大降低将来对以前功能做修改而产生副作用，以免对系统产生不利影响。在分布式环境下操作时，如需要更新组件，也不需要同时更新其他原有组件。

4. 每个子系统的编码可引入 Windows 2000 安全特性的优势，将在 12.8 节中说明。

5. 每个子系统必须验证或确保所有操作参量的正确性，这是通过在合适的点上捕捉其值和检查它的参量来达到的。通常这种探测在输出集上实施，特别是涉及到指针时。用这种

方法验证参量能确保防止调用者或它的一个线程在读操作前动态改变或者删除它的参数值。虽然好处是明显的,但由于参量经常送到寄存器,显式捕捉并不像它看上去那样简单。如果参数送入内存,它由系统服务调度器来探测和捕捉。

第三个设计目标是可移植性。分布式系统的一个重要的目标是在多种异构平台上进行操作,特别是希望系统的结构在许多不同的硬件平台上进行操作而只需最少量的重新编码的工作。这个好处不仅对用户,而且对微软也一样。因为可继续使用其资源,提高性能或开发新的产品,而不是在系统的最初版本发布后,仍在新的平台上重新进行开发。

第四个设计目标是关于性能的。为了达到高性能首先要通过选择好的算法和相应的数据结构,另外选择达到高性能必须不影响系统的灵活性。

第五个也是最后一个基于 Windows NT 技术的目标可分为两部分。首先,要求应用与 POSIX 相容的 API。与 POSIX 相容即允许应用能从一种平台(如 Windows 2000)移植到另一种平台(如 UNIX)。或者可能对于微软更为重要的是,从 UNIX 移植到 Windows 2000。做出 Windows 2000 与 POSIX 兼容的决策,包括 UNIX 风格的界面,可能是业界普及应用的一个里程碑决策。实际上一个分布式应用试图在非 POSIX 兼容系统上运行,必须处理各种 API。而在 POSIX 兼容环境中的相同应用,并不面向这样的异构性,从而能大大地简化。当然,一般认为这样的简化也可减少出错的概率。另外,一些政府合同要求与 POSIX 的兼容性。故在实际处于 Windows NT 执行体外部的 C/S 结构中,实际子系统的设计将 OS/2 和 POSIX API 作为被保护子系统来实现。子系统的设计采用了基于 Windows NT 的技术,与在运行中包含 API 的初始 NT 设计版本不同,它能支持所有的 Windows 2000 设计目标。

最后目标的第二部分涉及政府要求的 C2 安全认证。该安全认证要求系统包含安全性能,例如审核能力,访问检测,基于每个用户的资源限额,以及资源保护。没有这些特性和 C2 安全认证,Windows 2000 就不能用于所有的政府部门。当然,某些商业应用不属于政府,因此不需采用 C2 安全认证系统,但大部分商业应用仍希望有这样的安全级别。

12.2 内核模式综述

正如第 2 章指出的,Windows 2000 的实现使用了微内核设计。第 2 章集中讨论了内核设计,Windows NT 有关内核设计的许多特性在 12.1 节中给出。这些信息的综述列于表 12-1 中。图 12-1 描述了 Windows NT 3.51 及其以前版本关于这些服务的组织,而图 12-2 则说明了 Windows NT 4.0 的服务组织,对此类架构的主要改进包含了作为用户模式服务的即插即用的实现。Windows 2000 的架构在图 12-3 中给出。如前所述,Windows 2000 有一内核,逻辑上分成两部分:执行体,将在 12.2.4 中进一步说明;使用内核对象,硬件的抽象层和设备驱动程序,将在 12.2.1 节到 12.2.3 节中分别叙述。

表 12-1 Windows NT 微内核

微内核共性	基于 Windows NT 技术的实现
模块化设计	模块化设计
容易维护	所希望的易维护性
区分所有依赖于结构的特性	使用分离的子系统
有限的设备管理	决定采用即插即用手段
容易扩充	建立一个易扩充的系统

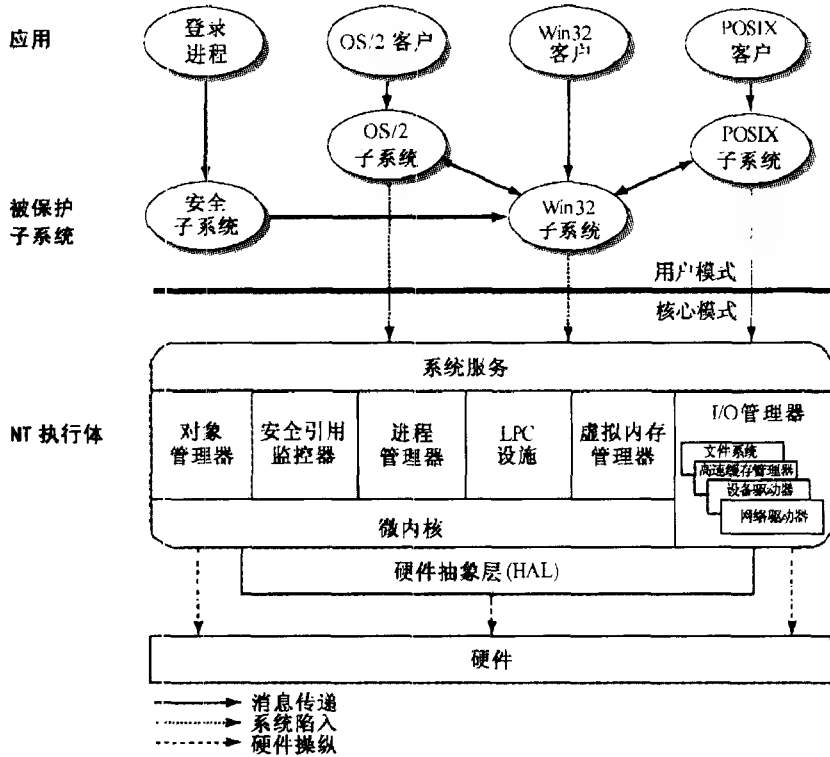


图 12-1 Windows NT 3.51 (及其早期版本) 基本体系结构

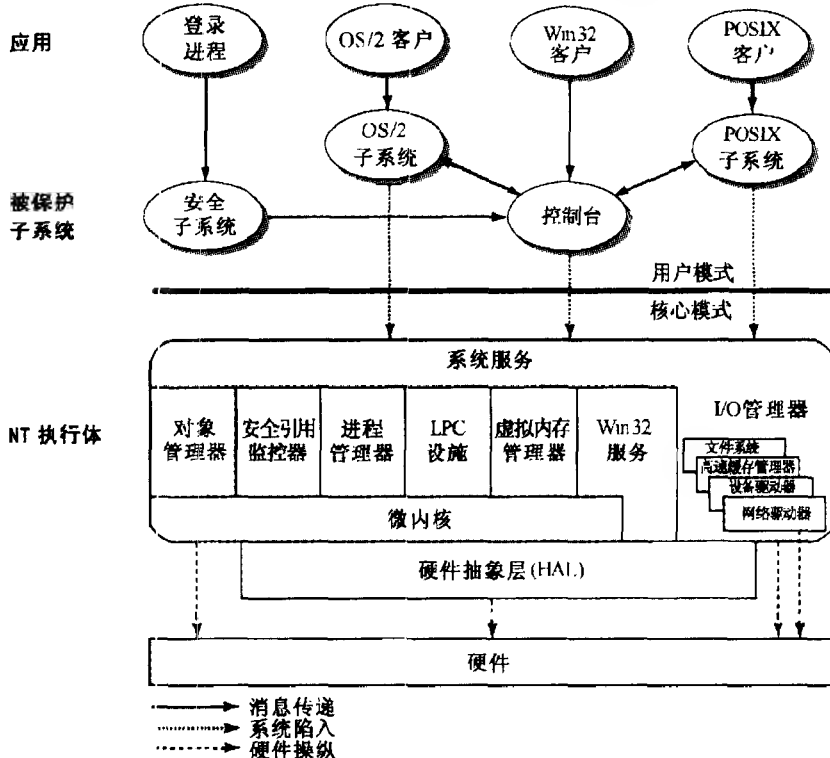


图 12-2 Windows NT 4.0 基本体系结构

12.2.1 内核对象

内核对象是一个简单对象集,支持创建执行体层的对象。内核对象按功能分成若干组。控制对象组包含以下对象:

- ◆ 内核进程对象。
- ◆ 延迟过程调用对象 (DPC)。
- ◆ 中断对象。

另一组对象称为调度对象,能改变或影响线程的处理,包含以下对象:

- ◆ 内核线程对象。
- ◆ 互斥体对象。
- ◆ 事件对象。
- ◆ 内核事件对象。
- ◆ 信号量对象。
- ◆ 计时器对象。
- ◆ 能延迟的计时器对象。

295

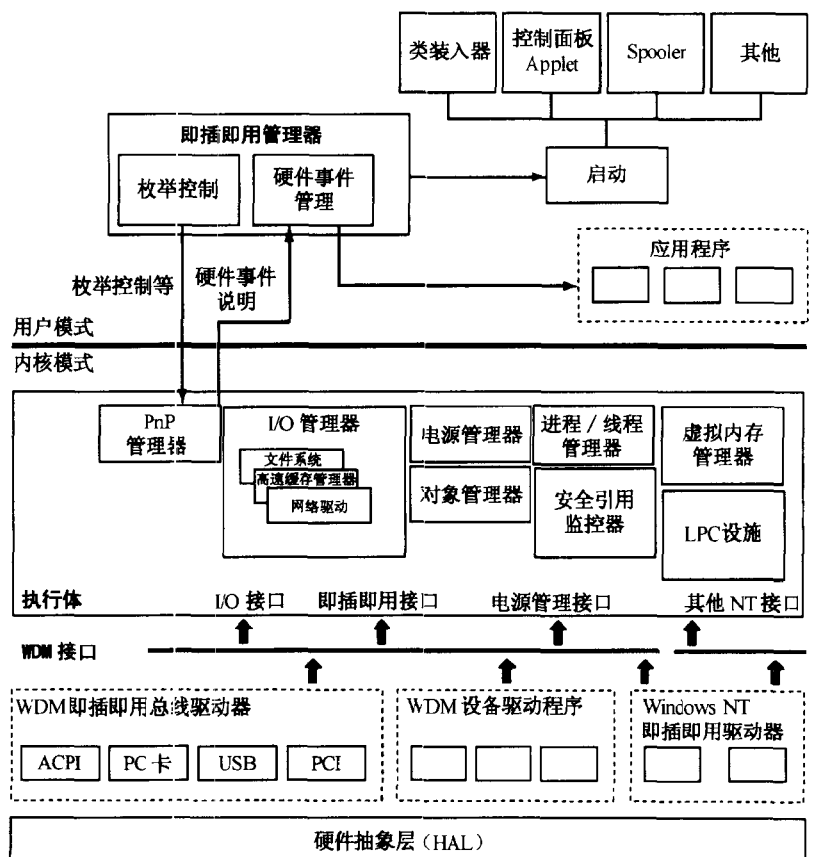


图 12-3 Windows 2000 即插即用体系结构

这些对象提供了定义良好的、能预测的原语和机制,当与执行体结合使用时,这些对象

296 能建立更多的高级机制。在这一层实现的惟一策略包括线程的调度和分派，而其他的策略留在执行体内，因此允许设计的最大灵活性。

12.2.2 硬件抽象层

硬件抽象层（HAL）是可装载的核心模式模块，这一层提供了 Windows 2000 运行的实际物理硬件的接口，功能是与特定体系结构相关的，如线程上下文转换是在 Windows 2000 HAL 中实现的。所有依赖于硬件的细节，如 I/O 接口与中断控制器等，均在该层中实现。而其他 Windows 2000 内部组件与应用程序都与 HAL 相联系，以获得依赖于平台的信息与功能。这样的设计允许 HAL 以上系统的每一部分能满足异构分布式环境需要的硬件独立操作，并能够维持可移植性。

12.2.3 设备驱动程序

设备驱动程序是可加载的核心模式模块。大多数设备驱动程序以带有扩展名“SYS”的形式存储文件。这些驱动程序提供了执行体的 I/O 系统与 HAL 间的接口。如果实现正确，设备驱动程序总是调用 HAL 程序，故而维护了整个系统的可移植性。Windows 2000 中设备管理和抽象的整个方法是为了协调和支持即插即用而继续改进 NT 4.0 的一个方面，鉴于它对 Windows 2000 的重要性，设备驱动程序以及即插即用的更详细的信息将在 12.3 节中叙述。

12.2.4 执行程序

执行程序实现 Windows 2000 的系统服务，要求所有的系统服务能确保它的所有参数的有效性。通常，所有参数在适当点上被捕获和检测，特别是对于指针参数和需要检测访问权的参数。这些措施能防止调用者或调用者线程在参数值验证后再去动态地改变它们。这看起来似乎加重了系统的负担，但大部分参数是送入寄存器，并不需要显式地捕获。系统服务操作的一个算法在详细说明 12.1 中进行描述。

297

详细说明 12.1

系统服务操作

调用一个系统服务时，按下述步骤进行：

1. 陷入处理程序得到控制权。
2. 保存状态。
3. 将控制权传送给系统服务分派程序。
4. 系统服务分派程序获得对应系统服务的地址。
5. 系统服务决定内存中参数的数目（从分派表获得）。
6. 如原先是核心模式，内核调用该服务，参数不会是坏的。如原先是用户模式，则地址被检测以确保访问是在用户空间内。如发生非法访问，就引起一个例外。

所有执行程序对象是由一组内核对象组成。下面是 Windows 2000 核心模式系统服务的列表：

1. 对象管理器: 对象管理器创建、管理和删除 Windows 2000 执行程序对象, 与相关的内核对象相比, 执行程序有附加的语义和功能。

2. 进程和线程管理器: 进程和线程管理器创建和终止所有的进程和线程。进程和线程管理器有时可考虑为一种对象管理器。线程作为一个专题已在第 2 章讨论过了。

3. 安全引用监控器: 该监控器负责在本地计算机上实施所有的安全策略。该职责可扩充资源并实施运行时的对象保护和审查。

4. 局部进程调用 (Local procedure call, LPC) 设施: LPC 设施的职责是在客户进程和服务进程之间传送消息。它是一种灵活的、优化的、标准的 RPC 版本, 相关内容已在第四章中讨论过。

5. 虚拟内存管理器: 虚拟内存管理器不仅是实现虚拟内存, 也负责对有关私有地址空间实施内存管理调度, 保护上述空间, 并支持底层高速缓存。该地址空间在 Windows 2000 中的布局在详细说明 12.2 中进行描述。

6. I/O 管理器: I/O 管理器将所有独立设备的 I/O 请求放到相应设备驱动器上去实现。I/O 管理程序与文件系统、高速缓存管理器、设备驱动程序和网络驱动程序协同工作。

298

详细说明 12.2 地址空间布局

作为内核设计的一部分, 地址空间的布局设计确保了用户地址空间和系统地址空间的分离。在这两个区域之间有 64 K 内存的隔栏, 内核和所有用户都不能访问该内存区域, 只允许简单地与边界比较, 以确定是合法地址, 防止越界。内核和核心模式占用系统区所有的页。用户拥有用户地址空间的页, 执行时绝不会在用户地址空间创建由内核占有的页。相反, 如有一页既映射到用户又映射到内核, 则该页仍应保存在内核地址空间。

Windows 2000 相对 NT 4.0 的另一个重大改进和变动是支持海量内存 (very large memory, VLM)。VLM 能支持 28GB 的内存和 64 位的 NT 版本。所有 x86 系列的处理器都不能支持 VLM, 但 Intel 与 HP 协作开发的一种新的结构系列 (名为 Merced) 即将推出, 适用于真正的 64 位 Windows 2000。真正的 64 位系统意味着每个 64 位的进程将有一个独立的, 巨大的地址空间, 尺寸至少是 512GB。这样就能满足当今和将来基于微软的应用程序对增加存储容量和处理的要求。一种新的 Win32 API 的小集合能应用于执行 Win32 或 Win64 二进制代码。只有能提供 VLM 的 Win32 应用程序能完全利用 64 位指针的优势。任何通过 64 位指针访问的地址必须返回上锁的物理内存。虚拟内存不能提交给应用程序, 因为不能物理地提供。缺页错误在 VLM 中不会发生, 除非那些地址是用来将数据映射到一个实际的物理文件。尽管有这些限制, 当前的实现仍被认为是朝着未来正确方向而迈出的重要一步, 将对类似于数据库管理那样数据集中的应用带来好处。

12.3 即插即用

即插即用结合了软、硬件的支持, 提供了微软 Windows 2000 (半) 自动识别和适应硬件配置改动的能力, 从而允许设备动态地变化, 尤其是这些改变不需要手工配置, 或是对设备

299

有具体的了解，而是由即插即用管理器自动地接收设备配置。即插即用不仅能让动态改变的一组设备一起工作，而且还能不需要用户的介入。有许多理由可以证明此特性对分布式系统很有利。一个实际的好处是减轻系统管理员的负担，因为有许多设备和系统接入分布式系统。系统管理员的注意力应该放到一些细节上，而不是在每个成员系统加入了一个新的部件时去配置系统。此外，即插即用对于笔记本电脑来讲也允许某一时刻连接到局域网，然后关闭，再经调制解调器与另一局域网连接，而不需要用户参与对系统的配置。如果说即插即用对 Windows 并不新鲜，但对于基于 NT 技术的产品线和 Windows 2000 来说则是新的。即插即用的简短历史在详细说明 12.3 中叙述。

详细说明 12.3

即插即用历史

即插即用首先在微软的 Windows 95 中引入，它的目的是简化个人计算机终端用户的使用。Windows 95 的即插即用版本依赖于高级电源管理 (APM) BIOS 或者即插即用的 BIOS。两个 BIOS 版本不仅支持即插即用，也支持电源管理。从此时起，APM 由高级配置和电源接口 (ACPI) 替代，允许操作系统控制电源管理和新的配置能力。Windows 98 仍支持 APM，目的是为了向后具有兼容性。

即插即用设计是由驱动程序及对应的对象组成的分层结构构成，下面讨论这种结构中的每一种主要部件。

核心模式即插即用管理器负责维护中心控制。具体地说，由它指挥总线驱动器执行所连接设备的枚举和配置，接着引导设备驱动程序加入一个设备，启动该设备或执行其他任何相应的操作。管理器必须与用户模式即插即用中的对应部分配合以协调设备的暂停、移走和同步，这些设备在系统认为需要且合适的时间应该可以提供这些操作。

核心模式组件中，电源管理器与策略管理器协同处理电源管理 API、协同电源事件以及生成有关电源管理的 I/O 请求包 (IRP)。例如，当系统同时接收到多个“关闭”请求时，由电源管理器收集这些请求。当收到请求后，它决定是否所有请求需要序列化或排序，并生成相应的 IRP，策略管理器监控系统的动作。收集的信息涉及到用户、应用程序和设备驱动程序的状态，把它们集成到电源策略中。在请求或特殊情况下，策略管理器也能生成 IRP 来改变设备电源状态。

300

I/O 管理器提供设备驱动的核心服务，具体地讲就是将用户模式的读或写翻译成读或写的 IRP 的核心模式组件。另外，它还管理所有的主系统 IRP。Windows 2000 中的 I/O 管理器的接口与 Windows NT 4.0 的相同，因此尽管 4.0 并不支持即插即用，也可在 Windows NT 4.0 上配置一个即插即用驱动器。

微软的 Win32 驱动器模型 (Win32 Driver Model, WDM) 的即插即用接口为驱动器提供了一个分层的结构。该接口允许不同类型的 WDM 驱动器、驱动器层和设备对象。在即插即用结构中，有三种基本类型的驱动器。

1. 总线驱动器：驱动 I/O 总线与任何独立于设备的每个槽口功能。即插即用总线是连接物理、逻辑或虚拟设备等的设备，可包含传统总线如 SCSI 或 PCI，也可以是并行或 RS232 串

行端口的总线。

2. 功能驱动器: 驱动个别的设备。

3. 过滤驱动器: 通过过滤对某个设备、一类设备或某一总线的请求, 隔离对设备的 I/O 请求。

每一设备能有两个或更多的驱动器层——有关的底层 I/O 总线的总线驱动器, 明确的即插即用管理器以及某设备的功能驱动器。换句话说, 某一给定设备对总线和/或设备可使用一个或多个过滤驱动器。

每个即插即用驱动器为每一个由其控制的设备建立如下的设备对象:

1. 物理设备对象 (PDO): 表示总线驱动器的总线上的独立设备。
2. 功能设备对象 (FDO): 表示功能驱动器上的独立设备。
3. 过滤设备对象 (过滤 DO): 表示过滤驱动器上的独立设备。

WDM 总线驱动器用来控制总线电源管理以及即插即用。WDM 总线驱动器是标准的 WDM 驱动器, 用来扩充总线的能力。回顾在 Windows 2000 中, 有着对其他设备的枚举的任一设备可看作是一个总线。下面是 WDM 总线驱动器基本职责:

- ◆ 在总线上枚举设备, 应包括标识这些设备和创建设备的设备对象。
- ◆ 报告系统在其总线上的动态事件。
- ◆ 响应 IRP。
- ◆ 合适时多路访问总线。
- ◆ 执行其他所需的工作, 以管理总线上的设备。

每当总线驱动器以设备名义在其总线上实行某些操作时, 它并不对设备执行读/写操作 (实际上是由有关的功能驱动器来执行)。惟一的例外情况是当总线驱动器合并到与功能驱动器一样的执行总线驱动器职责的驱动器中。作为这种多功能驱动器的一个例子是包含总线控制器、适配器、网桥或是 ACPI 驱动器的总线驱动器。关于 ACPI 见详细说明 12.4。

301

详细说明 12.4

高级配置与电源接口

ACPI 是由 OneNow 设计思路而来的, 它希望定义一种综合的系统范围的方法来控制系统和配置设备以及管理电源 [Micr98]。ACPI 1.0 版本定义了一种新的系统板和 BIOS, 该设计的目的是扩充即插即用数据使其包含电源管理和附加配置的能力, 所有这些是在操作系统的完全控制之下。

Windows 2000 对包含了 ACPI 系统板的计算机进行了优化。ACPI 的主要好处之一是提供给操作系统获悉某种设备的能力, 此种设备既不能以标准方法枚举给总线驱动器, 也可能只是一个新的设备 (例如一个嵌入式控制器)。此外, ACPI 能装入高层过滤驱动器。这种能力对于访问那些功能超过了总线标准的设备是非常有用的, 因为它可以提供这种访问功能。例如, 有人装入了带有电源控制的图形控制器, 但 PCI 总线不支持, 而 ACPI 驱动器可装入高层过滤器访问这些附加的功能。使用这种方法, ACPI 能为系统提供附加的灵活性。这种设计不仅给需要高级电源管理的膝上型电脑提供了好处, 对台式计算机用户也是很有益的, 他们不再需要等待操作系统的启动。

用户模式即插即用组件提供用户模式 API，用来控制和配置设备，它们是基于 Windows 95 的 32 位扩充版本配置管理器 API。在功能上这些 APIs 是 Windows 95 Setupxdi 程序的一个超集，但仅仅是用户模式 API，其应用程序能用于创建定制的硬件事件管理或是建立新硬件事件。

上面叙述了体系结构，现在讨论即插即用在 Windows 2000 的串行设备 RS232 中的实现。Windows 的早期版本要求每个硬件供应商实现检测协议和提供总线驱动器，在 Windows 2000 中由 Serenum.sys（表示串行枚举，Serial enumeration）提供这些功能。当 Serenum.sys 找到一个硬件，就为发现的设备建立一个 PDO。Serenum.sys 既发挥了 RS-232 设备的总线驱动器功能，也发挥了过滤驱动器的功能。由于 Windows 2000 已实现了协议和驱动器，上述设计能享用所有重用代码的传统优势。由 Serenum.sys 处理的传统串行设备包括鼠标、指示设备、调制解调器、数字相机等。所有非即插即用串行设备不使用 Serenum.sys，它们需人工安装。这些遗留的设备和它们遗留的驱动器在即插即用环境中与在原遗留环境中进行同样的工作。实际上有些即插即用设备的安装过程并不选择利用即插即用驱动器，因此它们并不享用优化的即插即用支持。有关微软 Windows NT 4.0 如何处理串行设备见详细说明 12.5。

详细说明 12.5

Windows NT 4.0 串行设备驱动器

为了允许独立硬件供应商在 NT 4.0 上使用串行总线驱动器，需要按一定次序装上所有串行设备驱动程序。串行总线驱动器叫做 Serial.sys，允许硬件连接到 NT 4.0 的计算机上。为 4.0 设计的驱动器依赖于 Serial.sys 的装入次序，需要经过修改才能在 Windows 2000 中操作。

12.4 Windows 2000 中的 NT 文件系统

Windows NTFS 可参阅 [CAPK98, Micr98]。它支持几种文件系统，包括 NT 文件系统 (NTFS)、文件分配表 (FAT 用于 3.5 英寸磁盘中)、FAT 32 (用于 Windows 98 卷)、CD 文件系统 (CDFS, 用于存储在光盘上的数据) 和通用磁盘格式 (UDF, 用于按 DVD 格式存储的数据)。文件系统如能提供文件在不同介质之间的替换则认为是兼容的。所有前面提到的文件系统除了 NTFS 以外都是兼容的。然而，在 Win32 中调用文件系统操作不考虑其底层的文件系统类型，给定卷可使用任何文件系统，默认文件系统的类型是在卷被格式化时就确定的。对于 NTFS 第 5 版，底层的盘上数据格式已改变过，需要所有 NT 4.0 和使用 Windows 2000 和 NT 4.0 的组合的系统使用一个服务包以实现互操作。新的 NTFS 5.0 已有称为面向卷或面向目录的特性。新的 NTFS 5.0 面向卷的特性使访问控制表 (ACL) 的检索加速，将在 12.4.1 节中说明。Windows 2000 中新 NTFS 面向文件的特性包括再解析点，将在 12.4.2 节中说明。还有优化的存储管理，这将在 12.4.3 节中叙述。附加的特性，如修改日志和复制支持都与目录服务相关，它们将在 12.5 节中讨论。

12.4.1 访问控制表

NT 4.0 采用访问控制表 (ACL) 作为安全措施的一部分来保护访问。文件 ACL 是在每次

物理共享时进行管理,而不能在系统范围内进行管理,另外 ACL 不能维持不同卷之间的一致性。由于 ACL 不能在不同卷之间保持一致性,最后的这一个事实可能会使人们吃惊,但这一决策是必需的,其理由如下:

- ◆ 为了使 ACL 能保证在不同卷之间状态一致,必须采用集中式数据库。该数据库可以“网络使用”或通过网络使用物理资源,从而绕过了 ACL 系统。
- ◆ 由于逻辑分布式文件系统在 FAT 和 NTFS 卷间通用,没有实用的方法能设置一个从 NTFS 继承的 DENY ACL,使其从 FAT 传至 NTFS,终止于 NetWare 卷(很可能出现要求托管适当功能的需求情况)。
- ◆ 为了建立一个工具以“走查”逻辑名字空间以及合适地设置 ACL,需要一种复杂的组合了消息/事务的引擎来对不可靠网络进行排队和更新。这种独特的想法在很多情况下是很危险的,包括需要资源去执行。
- ◆ 由于 Windows 2000 支持存储限额,卷间 ACL 的支持需要存储和计算所有可能卷的限额。

304

要支持所有前面提到的要求,必须有一种“全有或全无”选择。“全有”选择会产生一个非常慢的系统,因此如果依靠当今的技术它是不可行的。或许随着技术的继续改进,某一天此技术能在 Windows 环境中提供。

Windows 2000 中,加入了称为 ACL 检索加速器的优化,可使 ACL 更高效,尤其是每个 ACL 在给定卷中只会物理地出现一次。可以采用 hash 函数以及将所有 ACL 存储在公共区域中。微软 [Micr98] 指出采用这种选择的两大好处:首先,维护 ACL 需要的总的存储空间可大大减少;其次,由于负载的降低,验证和允许用户访问相应文件所需的总的时间也大为减少。但是,当前还没有可比较的测试标准。

12.4.2 再解析点

为了允许任意地操作一个文件或目录,在 Windows 2000 中采用了再解析点,它能使用分层的文件系统过滤器。这非常有利于处理从一个系统传送到另一系统的文件,或是几个不同类型的文件合并到一个文件中。相对于严格按标准方法来确定的静态处理(参见第 8 章),这种技术允许在处理的时候才确定如何处理文件,由此可建立一类动态文件处理能力。实际上,这样可以防止系统“按封面来评判一本书”的弊病,而提供了再解析和重新评估应如何通过使用过滤器来处理文件的能力。通过强制文件名再解析,再解析点改变了标准打开文件处理的方式。每当遇到一个再解析点,NTFS 将再解析数据放置到相应缓冲区中,该缓冲区可以由文件系统过滤器接受。再解析点是一个系统控制的属性,称为 \$REPARSE。它大大改善了系统处理不同类型文件的能力,因为在分布式环境中,由于系统的存在,有些文件类型可能是非本地的。任何文件或空目录都可设置该属性。再解析标记是 32 位长,分成下列几个组:

- ◆ 0—15 位:标记值。当文件需要再解析时,该字段用来确定专门的再解析标记。
- ◆ 16—28 位:保留位。该字段用来与标记值联结。
- ◆ 29 位: N。第三位是名字代用指示位。如果设置成 1,名字代用位指出:再解析点表示系统中另一个命名实体。
- ◆ 30 位: L。第二位是延迟位,如果设置成 1,该延迟位表示将使用足够的延迟去取出

305

第一个字节，并由此导致用户界面需用一个图标显示该实体，例如一个沙漏。

- ◆ 31 位：最高次序位是微软位。如果再解析标记是微软标记，就设置成 1，否则就设置成 0。

如果遇到一个再解析点，用户控制的数据将放入缓冲区，并使它对文件系统可用。通过包含在保留位和标记值中的信息，安装好的文件过滤器会来处理该请求。每个过滤器有两次机会检查每个 I/O 请求，就是在请求进出过滤器堆栈的时候。NTFS 自身放在该堆栈中，在其他的过滤器之下，再解析点的主要优越性是能使 Windows 2000 与第三方文件系统过滤器相互操作，也允许实施定制的和高级的存储管理特性。

12.4.3 存储管理

基于服务器的以及远程存储环境的有效存储管理是由 Windows 2000 中的两个服务提供的。第一个存储管理服务是一种单实例存储方案，该方法可识别在一卷内复制的数据流，并且可用单一实例的引用替换。经常无意间被复制的信息包括字体文件、共享库或是能执行的共享应用。如果这些共享数据已被修改，引用应按写时复制方法或是按第 8 章中描述的事务完成方法标注。正如人们所设想的那样，单实例存储方法的优点包括减少存储空间和简化管理机制。

第二个调度方法是远程存储，这种机制允许需要在更新时调用远程存储成为本地存储。通常远程存储使用的数据是曾经活动的数据，但此后成为相对的非活动数据。远程存储的逻辑非常类似于储藏冬天的外套。在冬天，外套是活动的，经常要使用，但当春回大地时，这些外套变成不经常使用，会搬到次要的存储位置，例如雪松木箱中。设想突然出现一个寒流（当然，是在外套移至木箱后的一周），如果当你打开衣柜，该外套已意想不到地从木箱回到衣柜而不需你的物理介入，那将是何等精彩，最多只需要打开衣柜门并等待一会儿。本例展示了远程存储调度方法的行为。数据移至远程存储后仍出现在本地，不过搜寻该数据的延迟可能会稍微增加。最明显的好处是用户的操作不需要改变，而且用户不需要知道数据已经从主存储器移至辅助存储器或是相反，这些操作对用户是透明的。

306

12.5 活动目录

Windows 2000 服务器的重大变化之一是引入了新设计的目录服务即所谓的活动目录，它吸取了第 8 章中叙述的 DNS 和 X.500 的最佳特性的优点。这种服务大大改善了以前的分布式文件系统（DFS）服务，DFS 是微软基于 Windows NT 技术的早期版本。活动目录为当前的 Windows NT 4.0 操作系统用户提供的易于升级的机制。微软将活动目录描述为一种安全的、分布式的、分割的和复制的目录服务。它的作用是简化分布环境下信息的导航和管理。它提供了简单明了的 API，支持广泛的定义良好的协议和格式。它与 X.500 的关系在详细说明 12.6 中有相关描述。微软是希望设计成一种提供透明的，但紧密集成的目录系统以帮助管理和维护当今网络计算环境下的海量文件和资源（如果历史是简单地重复，这些可能对于明天的存储需求来讲太小了）。建立活动目录是为了解决分布式环境中多种服务器等级与名字空间的统一和排序问题，时间将验证它是否解决了这些问题。活动目录的设计特性和优点如下：

- ◆ API 设计成便于脚本语言和 C/C++ 的程序员使用的形式。

详细说明 12.6

活动目录与 X.500

由于不需要系统将第 8 章中叙述的 X.500 完整的目录系统作为宿主, Windows 2000 通过使用在 RFC 2251, RFC 2252, RFC 2255 和 RFC 2256 中提到的轻量级目录访问协议 (LDAP) 来支持 X.500 信息模型。另外, 活动目录支持 X.500 的目录访问协议 (DAP), 目录系统协议 (DSP) 和目录信息隐藏协议 (DISP)。微软 Windows 2000 操作系统方案不同于 X.500, 主要有以下几点:

1. Windows 2000 基于 TCP/IP, 而不是 X.500 预期的 OSI。
2. X.500 需要一些定义过的部门来管理树的顶层, 即国际上每个国家的全球目录服务结构。Windows 2000 不使用这样一种全球结构。
3. X.500 中的名字是一系列的三元组, 如第 8 章所展示的, 而 Windows 2000 使用 Internet 格式的名字, 如 galli@companyname.com
4. Windows 2000 定义了实际的 API, 以允许各种工具与它的目录服务通过接口进行通信; X.500 不定义 API。

- ◆ 使用拖放式管理和分层域结构, 目的是使管理者感到既简单又直观。
- ◆ 通过一种可扩展的调度方法提供目录对象的可扩展性。
- ◆ 通过一种全局目录快速查找和 Internet 发布。
- ◆ 利用多主体复制来获得有效和方便的更新。
- ◆ 支持短周期跨度的服务, 如图表, IP 电话和会议服务。
- ◆ 向后兼容性。
- ◆ 与 NetWare 环境的互操作。

活动目录可看做是服务提供者, 它的功能汇总在图 12-4。目录结构可跟据使用该系统的

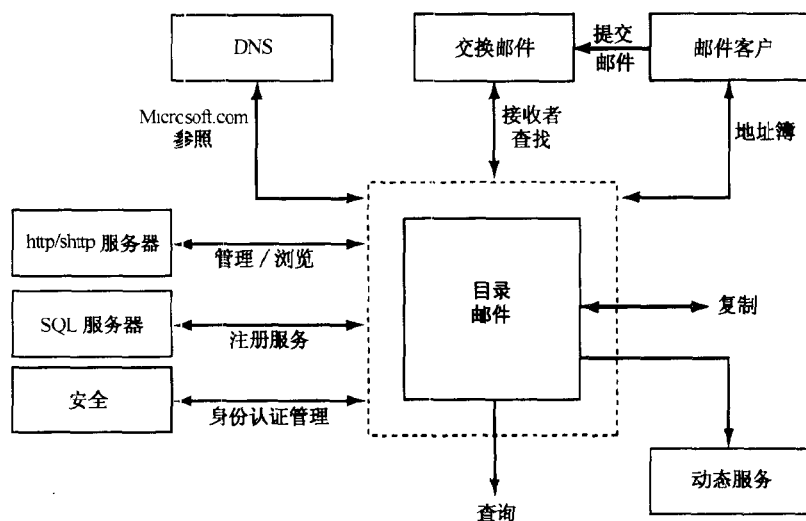


图 12-4 活动目录功能图

307 企业的规模大小而进行灵活设计，因此它能满足从最小的公司到最大的跨国公司对目录的需求。当然，最大的挑战是能提供整个系统足够的稳定性以使人们信服。无论如何，微软承诺：该目录系统不仅能与 Windows NT 服务器的早期版本互操作，也能与大部分 Novel NetWare 3.x 和 4.x 实现互操作。

12.5.1 名字空间

Internet 名字空间的概念通过使用 DNS 作为本地服务而集成到 Windows 2000 目录服务概念中。采用这个本地服务将一个名字翻译成 IP 地址。这种集成的意图是建立一个可扩展的、统一的、可管理的环境。这种名字空间的优点如下：

- ◆ 支持开放的标准，包括域名系统（DNS）和标准协议，例如 LDAP 和 HTTP。有关名字空间的更多信息已在 12.4.1 节中给出。
- ◆ 支持标准的名字格式以便于迁移和使用。

Windows 2000 允许多重域名连接成树结构。一个分别从本地和经 Internet 观察到的有关对象名的例子如图 12-5 所示。Windows 2000 并不使用主要的或备用的域名控制器，而是认为所有的域名控制器是平等的，因此任何以及所有对域名控制器的更新将复制到所有其他的域名控制器。为了解析不认识的名字，Windows 2000 操作系统采用 Windows Internet 名字服务。Windows Internet 名字服务与 DNS 的集成是一个过渡性产品直至 DNS 的 Internet 标准更新为支持动态 DNS 的时候为止。动态 DNS 允许客户端被动态地指派地址，并自由地注册和更新 DNS 表。

将多重域构成一棵域树是微软 Windows 2000 能提供可扩展性的主要原因。特别是名字空间结构利用自底向上的方法构建树，而自顶向下分割过程更复杂、更耗费时间。整个名字空间构成一片森林，认为每个子树是 Windows 2000 的一个域，并向 DNS 通报它的存在。这种树结构因活动目录而是可能的。每个域是目录的完整部分，域能进一步细分为组织单元（OU），如图 12-6 所示。这些 OU 用于管理目的，也额外地对建立域树中上千万个对象的结构有所帮助。

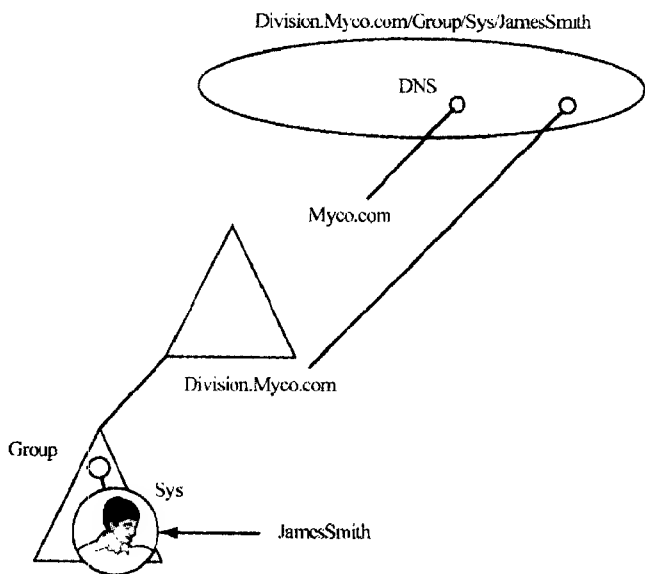


图 12-5 Windows 2000 基于 DNS 的位置服务的两种视图

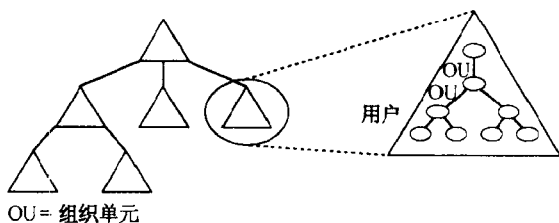


图 12-6 域划分成组织单元（OU）

12.5.2 通过修改日志实现复制和可扩展性

现在研究使微软的对等多主体复制机制取得最大效率的方法。在任何大型系统中要管理大量共享文件的机器集合,需要一种简单和合理的文件共享复制服务,如同第8章中讨论过的。本节中将说明 Windows 2000 提供的文件复制服务。我们在第4章中已学过存储管理的内容,如果文件严格地为读而进行复制,实现是很容易的。如果大多数的文件访问只是只读的,系统总的效率将保持最低。活动目录方法的关键是基于这样的观测:所有的目录服务需求中99%是询问的,而更新需求仅占1%。多主体复制方法允许多重复制,其中的每一个都被认为是主体,而不是相对一般的主/副方法。因此,这似乎是更先进的对等解决方案,对等方法显然具有更大的可扩展性。

310

可能会产生这样一个问题:在采用多主体拷贝时,如何维持第9章中所讨论的数据的一致性。为了获得数据变化的某种次序,Windows 2000 使用一种修改日志的方法。修改日志在卷上实现修正控制。修正控制是跟踪修改的通用概念,它是通过对每个改变加以标记记录并枚举每个改变。修改日志能使系统管理员管理大量的文件和目录,并跟踪这些实体的变动。每次新的改变都附有改变的理由,这些改变理由都伴随着打开/关闭对话。微软通过采用更新序列数(USN)和缓和使用物理时间戳排序引起的所有问题(在第10章中讨论)来选择使用逻辑排序。这样,每个记录包含64位USN标识。USN随着每次修改而单调上升。当没有记录维持只读访问时,有21个操作会引起修改记录,这些操作如下:

- ◆ 建立一个目录。
- ◆ 删除一个目录。
- ◆ 修改文件中的数据。
- ◆ 向文件中附加数据。
- ◆ 修改文件属性,包括ACL。
- ◆ 重命名一个文件。

每个这种修改记录通常小于128字节。在这128字节中,USN记录包含指向父文件、源文件、改变的实体以及提示改变原因的域的引用。通过使用修改日志,Windows 2000 允许信息的多重存储,允许每个存储保存多于1千万个对象。任何时候当用户在目录中写入一个对象,该对象就接收一个USN。USN按计算机保存,每次对象改变使其增加。增加的值随后连同写入修改的计算机的签名一起写入该对象。除了利用USN作为对象内容,USN还集成到每个目录对象特性中。与对象USN类似,对象特性USN在每次特性修改时增加。复制服务与修改日志一起记录文件的修改并唯一地标识每次修改。这就提出了对于一个给定文件标识每次修改的能力的问题。

311

回顾一下USN记录仅当修改发生在磁盘上时才建立。因此,USN记录只跟踪持久的修改,这些修改只是在文件一级而不是卷一级被记录,通过一个字段来记录实际文件名,文件名长度,文件名偏移量以及文件名属性。这种方法的优点是记录数量直接与修改数量成比例,而不是卷中的条目数。旧的修改记录没有4k字节的存储单元。每个修改日志包含三部分的标识字段,必须不同于该实体原先的修改日志标识。第一个标识字段通常总是存放主版本号,第二部分记录次版本号。用版本2.01作为修改记录的例子,其主版本号是2,次版本号是01,这两部分标识能使每次改变都是唯一的标识。第三个也是最后的标识字段是安全

标识字段。USN 记录也包含一个时间戳值和源信息。源信息可以是以下三种中的一个：应用程序中的正常改变；存储功能（例如文件复制）引起的存储改变；由应用程序引起的辅助应用程序改变，例如为实现完整全面的图像的应用程序功能而需要产生的预览链接。

为了改变主拷贝之间的同步，复制成员请求源成员计算机将它所有大于最后接收到的 USN 的修改都予以更新。源计算机于是检查它的目录，并用大于当前 USN 的 USN 标识每个对象。特性修改是逐个地调节的。在冲突或是对同一特性的多重改变情况下，由时间戳来决定获胜者，即最近修改的获胜。由于时间戳只是用做平局时的仲裁者（因为没有硬币可抛掷），故同步不是重要的。

最后一点，前面提到的方法是基于数据的，而数据查询需要 99% 的时间。数据是经常改变的，并且是易失性的。易失性信息通常不存放到目录中，但需通过一种透明的联接机制连接，以替代存储在目录结构中的信息。也就是说，易失性对象存放在单独的采用不同复制特性的存储器中，但仍维持公共用户视图。

12.5.3 微软的索引服务器和 HTTP 支持

超文本传输协议（HTTP）是 Internet 的 World Wide Web（WWW）的标准。Windows 2000 的微软 Internet 信息服务器（IIS）将对目录对象的请求翻译成在任何 HTML 客户端都能看到的 HTML 页面。因此，任何询问和所有对象都能利用熟悉的 Web 浏览模型来实现。而且本地和远程资源差异是最小化的，因为这两种类型的访问是以同样的方法呈现在用户面前的。正如我们多次提到的：这是分布式操作系统的一个重要特性。IIS 的一个部分是微软索引服务器，它的职责是在企业 intranet 和 Internet 的站点上实现 Web 类型的浏览。索引服务器采用 12.4.1 节中描述的修改日志。在本节中，我们可以更仔细地观察索引服务器以及它与 Windows 2000 中目录的关系。

微软索引服务器允许一种数据库类型在一目录中访问对象，特别是它允许全文索引，这种基于文本的索引支持下列类型查找：

- ◆ 查找词或短语。
- ◆ 使用布尔操作符查找，例如 AND, OR 和 AND NOT。
- ◆ 使用统配符查找，例如 *, ? 和正规表达式。
- ◆ 使用关系限制符（<, <=, >, =>）查找诸如日期和文件尺寸那样的常量。
- ◆ 查找接近其他字的字。
- ◆ 在一种指定的文档类型中查找。
- ◆ 在指定类型文档中的指定段中查找。
- ◆ 基于特性的查找，例如作者和创建日期。
- ◆ 按匹配的质量排序。

或许微软索引服务器给人最深刻印象之一是该系统只需要零维护，它是自动更新的，并且包含自动故障检测和恢复。另外它的查找能力集成在系统安全中，因此查找只向允许访问的进行实例化查询的用户提供它的文件和信息。显然，任何热衷于 UNIX 的用户能完成任何一个或所有上述这些任务，但无法应用如此简单的界面。为了能以最低的开销提供这些服务，索引是在每个虚拟根的基础上被控制并支持索引的递增刷新。最后，它在国际环境下支持下面七种语言：

1. 英语。

2. 法语。
3. 德语。
4. 西班牙语。
5. 意大利语。
6. 荷兰语。
7. 瑞典语。

313

如果需要增加语言支持,其他语言可以很容易地通过使用开放的规范说明而集成进来。与有关其他类型语言相适应的规范说明现在还不清楚。详细说明 12.7 演示了索引过程。

详细说明 12.7

索引过程

索引服务器并不时常对修改作轮询而增加系统资源的负担,相反,这种服务器已在文件系统中注册以获得修改通知,并适时更新索引。索引过程是四步过程,第2步到第4步在图 12-7 中描述。

1. 收集改变通知:这是在后台工作的仅当资源可用时才发生的懒惰过程。当卷上的文档被修改,文件系统通知索引服务器。如果能使用资源,且不会影响系统的性能时,索引服务器就打开文档并开始索引过程。利用操作系统来收集修改通知要比传统的轮询方法具有更好的可扩展性。

2. 过滤:由于文档倾向于使用独有的文件格式,内容过滤器是由最熟悉这种特殊数据类型的程序员使用开放标准 IFilter 接口生成的。这样,微软为所有 Word 文档生成过滤器。过滤系统打开文件,随后选择并采用合适的内容过滤器。提取文本的语段并送到下一步。索引过滤器也能够处理嵌入对象,例如图形、电子表格,以及包含在其中的文本。最后,过滤过程能够为了传递信息对文本进行标记,例如文本分块所使用的语言。

3. 词分割:字符流从过滤过程输出必须分割成词,这是对计算机来说比对人类更加困难的任务。对于许多语言,例如英语,可利用空格来分割词,而其他语言,例如日文,就不是这样。但是,不论何种语言,字符流必须采用词分割来切分词。这些词是用 Unicode 表示和存储的。

4. 规格化:规格化过程用来清理词。规格化将处理大小写,标点符号和虚字,这样就可使词一旦要存放到索引中时就有统一的表示。通常虚字包含“无用的”或无意义的字,例如 the, of 和 and 等。虚字的省略能帮助减少索引的总长度,并使处理按规格定制。经规格化后词就放入文本索引中。

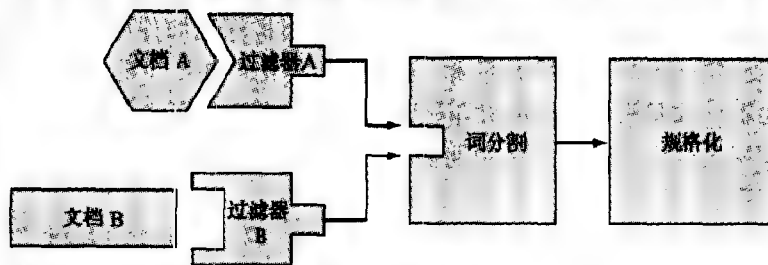


图 12-7 索引过程

12.6 微软管理控制台

内部代码命名为 Slate 的微软管理控制台 (MMC) 设计为 Windows NT 4.0 及以上版本的管理应用程序而提供的一个公用框架。当没有提供实际的任何管理应用程序时, MMC 提供一个公共环境和稳定的接口给“插件”管理应用程序, 并使它们无缝地集成到系统中, 这是 Windows 软件开发工具箱 (SDK) 的一部分。有了 MMC, 可以将多个较小的工具集合生成小型工具。每个工具就是所谓的插件, 因此可以方便地扩充和分层组织。最后形成的工具通常用来管理一类资源, 例如一个网络。当然, 每个建立的工具可存储起来以供将来使用和参考。而当允许建立多个工具时, 管理者只需要在某一时刻建立实际需要的工具, 可卸下当时不需要的任何其他的工具。它也可邮寄一个工具定义给其他管理者。当第二个管理者收到该工具的时候, 所有需要的插件会自动地下载和安装。

下面列出管理者使用 MMC 的四个好处:

1. 面向任务: MMC 中定义的工具是面向任务的。管理者可以使用来自不同厂商的工具来创建他们自己的工具。这些工具都是针对并显示了面向任务的信息, 而不是用未处理的对象来表示。

2. 集成: 利用单一控制台来收集由某一管理员必须执行的所有任务。用户接口将所有新增加的应用集成到已有的控制台上。

3. 委派: MMC 有能力使已存在的工具经修改后创建功能降低的并且复杂性也减低的新工具。这些较简单的工具能传送给其他的个人, 于是接收工具的个人可以完成他们的杂务, 即可以使用更简单、更易管理的工具完成他们的任务。

4. 接口简单化: 所有由 MMC 建立的工具具有相似的外观和感觉, 而无论开发这些工具的厂商是哪一家。从统计角度讲, 应该降低学习任何新的 MMC 工具的能力曲线。另外, 应能使用户选择或组合来自不同厂商的具有一致功能的 MMC 工具。后者是在建立和采用统一的 API 时具有的共同优点。

虽然对管理者是透明的, 所有 MMC 插件支持下面两种可能的模式中的一种或两种。

1. 独立插件: 这和类型的插件功能独立, 不需要任何类型的插件支持, 不依赖于任何的其他插件。

2. 扩充插件: 这和类型的插件只能在双亲插件调用它时才提供功能。它可作为一个专门类型插件的从属类型, 且每一次出现时均是此类型。

如果一个插件支持两种模式, 则它包含某些独立的功能, 它就可以独立工作, 同时也具有可选的功能, 扩展了已有的插件。如果已有的插件没有出现在给定环境中, 则扩充功能的子集 (也只有该子集) 是不能提供的。例如, 一个打印机标记插件可以在两种模式下工作。如果系统存有一张定义过的打印机的列表, 该打印机标记插件将在定义过的打印机上使用扩充模式工作。然而, 如果没有定义的打印机, 该插件就在独立模式下工作, 同时需要管理者输入定义的打印机集合, 这是该插件所期望的。详细说明 12.8 描述了可能使用的不同插件扩充类型的一些例子。

微软鼓励所有开发商使用 MMC 建立的一系列管理工具, 实际上它提供了用 MMC 的工具集去集成非 MMC 的工具的方法。尤其是可以在 .MSC 文件中建立并存储非 MMC 工具集的快捷方式。于是管理者能使用 MMC 的 .MSC 文件建立对这些非 MMC 工具的任何可执行程序的

详细说明 12.8**MMC 扩充类型**

下面是可用的 MMC 扩充类型的示例

1. **名字空间枚举**: 这种扩充类型在一个结点中操作。结点定义为一个可管理的任务或对象, 例如一台计算机或 Web 页的集合。名字空间是当前工具能提供的所有结点的排序列表。名字空间格式化成一棵树。
2. **工具条和工具条按钮**: 这些扩充类型是在视图中操作。视图定义为插件的可视表示。这些扩充可将整个工具条加入到窗口中, 或是在一个已有的工具条中加入一个按钮。
3. **新建**: 这种扩充类型用于给定的结点或对象, 将一些项目加入到“建立新菜单”结构中。允许对一给定结点多重建立“Create new”插件。

快捷方式。

12.7 集群服务

集群经常指多个独立系统的集合, 可发挥资源群协同工作的优点, 能比任何单独子系统提供更好的服务。为了达到实用的目的, 集群必须容易实现、编程和管理。理想情况下, 所有的客户端应用程序应能与集群交互, 并把它看做是一个简单的、独立的服务器。其优点是提高了可靠性和容错性。有时集群中的独立子系统要比非集群类型便宜得多, 所以整个集群的费用不像一些人想象的那么高。当然, 实际费用取决于选用怎么样的设备放入集群中。12.7.1 节给出了微软 Windows 2000 操作系统集群服务的概况, 它是 Windows 2000 服务器的一部分。12.7.2 节展示了不同集群的抽象, 12.7.3 节集中说明集群服务的体系结构, 12.7.4 节则论述应用领域的情况。

12.7.1 集群服务概况

微软的集群服务首先是在 Windows NT 4.0 操作系统中出现。它在微软的白皮书系列 [Micr98] 和 [GaShMa98] 中进行了说明。在 Windows 2000 的实现中包含了一种“向导”来帮助建立集群及其特性, 对基于第 6 章中叙述的 COM 和 DCOM 的已有服务或应用环境的集成提供了很大的改进。

集群服务设计成一种为可扩展的容错环境提供方便的接口。集群服务极大地依赖于 RPC 机制、Windows 2000 名字管理、网络接口管理、安全性、资源控制和文件系统。为了提供简化的接口, Windows 2000 集群服务采用了 12.7.2 节中描述的抽象。可扩展的容错结构在 12.7.3 节中说明。应用服务中能采用集群服务优点的方法将在 12.7.4 节中叙述。某些传统的集群特性包括容错的应用支持, 状态恢复和常量迁移等, 虽然当前在 Windows 2000 集群服务中还没有提供, 但已在 [GaShMa98] 中承诺, 希望在不远的将来能提供支持。

12.7.2 集群抽象

将 Windows 2000 操作系统中集群服务的抽象设计成有利于系统管理员、用户和集群服务自身。正如 [GaShMa98] 中描述的: Windows 2000 集群服务采用了一些抽象, 包括资源、资

源依赖性、资源组和虚拟 Windows 2000 服务器抽象。资源抽象是集群中管理的基本单元。有两种基本的预定义资源类型：物理资源和逻辑资源。物理资源的例子是一种小型的计算机系统接口（SCSI，发音为 scuzzy）磁盘，而作为逻辑资源的例子是一个 IP 地址。附加的资源类型及其相关的控制库（例如 DLL）可由应用开发者加到集群中，这样就能扩展集群的能力。这里不关心精确的资源类型，因为资源分类的目的是能使已有的应用最大程度地使用集群服务。主要是每个资源的接口必须是相同的，从而使所有资源易于监控和管理。

资源依赖性抽象利用一棵依赖树来描述，并定义集群中的初始服务序列，后者经常称为起始服务。例如，有一个实现对象代理的资源，就像 CORBA 环境中采用的一样，必须先于其他服务启动，而其他服务需要与对象代理通信并完成一个任务。该依赖性与下面的电话例子均采用同样的逻辑。具体地说，希望通信的各部分之间的连接必须在通信之前建立，这样会话就取决于已建立的通信连接。依赖树也用来说明和定义服务必须关闭的顺序，即通常称之为终止或关闭的顺序。依赖树也包含在集群中迁移一个资源到其他位置时必须维持的资源子集组所需要的有关信息。一个普通位置依赖性的例子是数据库和用来存储该数据库的磁盘。

资源组的抽象能使资源整合成较大的逻辑单元，这种抽象有利于资源管理。通常，将能执行应用程序的全部功能的相关资源整合在一起，组织成一个资源组。这样的组在各种方法中作为单一的单元来管理，如果组中的单个实体失效，那么整个组就会失效。另一种组可标识为失效—恢复组，这个新组必须能支持失效组中的每一个设备。精确的资源集和资源依赖树是由集群管理员按组指定。另外，管理员有责任制定失效—恢复策略，包括设置重新启动一个组的延迟时间以及何时服务应用会返回（作为失效—返回窗口），以及组的优先属主结点表等。

最后的抽象是虚拟 Windows 2000 服务器抽象，这种抽象将服务和提供给指定服务的所有需要的资源封装在一起。通常使用集群服务来配置的服务是文件服务器、数据库、Web 服务器、e-mail 服务器和其他标准的每日商务操作服务器等。对每一虚拟服务器来讲，网络名和 IP 地址对所有客户都是可见的。这些信息不因集群中的修改而改变，因而对所有客户、应用程序和管理者提供了一种单一稳定的环境感受。

12.7.3 集群服务体系结构

从物理方面来讲，一个集群可由经以太网连接成的个人计算机组成，也可由高性能超级计算机组成。无论哪种情况，集群提供了容错、可靠性和单一服务器或单一系统的功能。通过容错，集群服务能检测和重新启动失效的硬件和软件部件。如果一个失效部件不能恢复，该部件的功能就迁移到集群的另一个部件。失效—恢复组的备份部件提供了可靠性。不论是由于集群的需要或是由于失效—恢复操作引起的工作负载，通常对集群是透明的，于是进一步加深了单个服务器的感觉。

通常有两种软件模型用于集群技术：公共资源模型和独立资源模型。两种模型能在单一的 Windows 2000 集群中得到支持。一般来说，希望有最大的可扩展性的应用程序，应该采用独立资源模型。如果工作负载难于分散，则经常采用公共资源模型。现在来说明这两种模型。

公共资源模型允许一个资源中的系统去访问集群中与任何系统连接的任何资源，因此，集群中的所有资源都能相互访问。集群中公共资源的一个例子是系统磁盘，它可被集群中其

他的人访问。为了防止潜在的冲突,确保数据和操作的完整性,共享内存(见第 4 章)要遵循某种类型的并发控制(见第 5 章)和/或采用事务管理(见第 9 章)。

独立资源模型只允许一个系统在某一时刻拥有和访问一个给定的资源,除非拥有者失效。在拥有者失效的情况下,集群中的另一系统可以动态地成为拥有者。所有对集群中具体服务的请求会自动地送往所需资源的当前拥有者。如果某一客户端需要集群服务器执行一个需要访问多个资源的任务,选定集群中的一个部件为该请求的宿主,此宿主随后将该请求分解为多个子请求并做适当的分布处理,宿主再收集各部分的响应并汇总成集群的响应。

前面提到 Windows 2000 允许每个集群使用任一模型,或者两个模型一起使用。基本配置信息通常是保存在注册表——一个本地数据库中。在集群系统的情况下,采用集群范围的注册表是为了允许每个集群服务以及应用程序获得一个统一的关于资源状态和当前集群配置的全局视图。所有注册表的更新都是自动完成的,采用一种自动更新协议。

319

12.7.4 为应用程序配置的集群服务

有两种方法提供给采用基于集群服务的应用程序。第一种方法中应用知道服务是基于集群的服务,第二种方法不知道有集群存在。知道服务是使用基于集群的应用的方法经常称为惯例应用 [GaShMa98]。惯例应用并不需要修改应用,但要开发一个集群接口层,该层应用于管理和响应失效。因此,惯例应用可显式地获得集群自动失效—恢复的优越性,它通过惯用资源 DLL 检测故障。另外,这些接口提供了启动和停止集群中给定资源的能力,以及监控每个资源是否可操作。这样,对惯例应用控制的颗粒度的级别将大大增加。

不是所有应用都需要成为惯例应用,也可以是类属应用,这种类属应用不需要经历对集群服务操作的任何修改。惯例应用中控制和监控的颗粒度级别不会有很大变化,因此,可靠性和失效检测就比较原始了。失效检测不能在集群中的单个服务中实施,而通常通过检查进程的状态来实现。一个挂起的应用程序并不是一直可以检测到的,如果对类属应用需要更复杂的失效检测机制,则通常应实现一个惯用资源 DLL。

12.8 Windows 2000 安全性

正如第 11 章中讨论的,所有现代操作系统必须在许多层次上是安全的。Windows 2000 对下列的工业标准和安全协议提供了全方位的支持:

1. Diffie—Hellman 密钥交换。
2. 数字签名,专门的 hash 消息认证代码,例如 MD5, SHA 和 CBC。
3. Kerberos 认证协议。
4. SSL。
5. 私钥加密,专门的 DES。

除了这些工业标准以外,Windows 2000 还提供了三种其他的密钥安全特性,本章将研究这些特性。12.8.1 节研究安全配置编辑器,12.8.2 节给出加密文件系统,12.8.3 节说明微软安全支持提供者接口。

320

12.8.1 安全配置编辑器

微软安全配置编辑器是一种 MMC 插件工具。这种单独的工具设计成能帮助定义、管理

和验证基于 Windows 2000 的系统的配置，并且采用了成本-效益的方法。该工具在 Windows 2000 环境下特别有用，因为为了提高对企业级分布式计算的支持，管理安全配置的复杂性将大大增加。该工具并不意味着去替代用户管理、服务器管理和访问控制表编辑器，而是补充这些工具的功能。一般需要的操作应能自动地在后台执行。在表 12.2 中列出的安全特性，可通过安全配置编辑器来配置和分析。

安全配置编辑器的一个有用的特性是它定义和使用安全配置模板的能力。这些模板可包含安全属性的设置，能用于安全的任何方面，甚至管理者可对基于某一类基础模板的企业系统进行分析。当然，这样的基础模板既可以是 Windows 2000 中的默认模板集，也可以是由管理者为企业定义的用做推荐模板的模板集。这些模板是标准的基于文本的 .inf 文件。每当一个模板被安装，安全配置编辑器的配置引擎切分该模板并使系统作出相应的改变。通过该编辑器，管理者也能从一个已存在的系统创建一个模板。

表 12-2 SCE 配置的安全特性

安全性能	说 明
系统安全策略	该编辑器允许设置访问策略，包括如何和何时用户能登录系统、口令策略、整个系统对象安全、审查设置和域策略
用户账户	编辑器辅助设置组成员，特权以及用户权限
系统服务	编辑器能配置已安装的不同服务，包括网络、文件共享和打印
系统注册	编辑器能在系统注册表中采用一系列安全值
系统存储	编辑器能给本地系统文件卷和目录树设置安全功能
目录安全	编辑器可用来管理 Windows 2000 活动目录中对象的安全

321

有五种微软认定的基本安全领域。

1. 安全策略：这些策略适用于各种本地的和域的安全策略属性，包括机器层设置。Windows 2000 中的域定义为一种物理安全边界。策略包括可采用的口令类型、Kerberos 票据生存期、账户封锁等。

2. 受限组管理：该功能由系统管理员用在由敏感的成员构成的组中，一般的受限组包括打印机操作员，服务器操作员，电源用户等等。组的分析不仅依据简单的成员关系，而且可以是递归的成员关系。

3. 权限和特权管理：该领域属于用户和组的管理，用户和组的管理授予系统中专门的权限和特权。该领域包括指定可信任域和组的能力，它们也可在本地机器上享有权限和特权。

4. 对象树：该领域包括目录对象，注册键和本地文件系统。安全编辑器允许配置对象拥有关系，访问控制表以及审查信息等。对于目录对象，管理者可以利用安全描述定义语言 (SDDL)，它用来完成具有资格的 LDAP 名字列表和每个名字的安全描述符的完全说明。有资格的完整注册键路径和使用 SDDL 格式的安全描述符可用来控制注册表对象。文件系统安全性采用动态继承模型，NTFS 也支持此模型。给定系统的所有卷作为单一的树处理。

5. 系统服务：该领域包括所有本地和网络系统服务。这个领域设计成可扩充的，可以包含诸如文件和打印服务，电话和传真服务以及 Internet/intranet 服务的各类服务，此类目录中的最重要的限制是在该安全配置编辑器中使用服务的名字，需要与服务控制管理器中使用的一致。

额外的领域可通过 MMC 插件加入, 不需要打破与已有模板的向后兼容性。对每一个这样的领域, 系统管理员能通过图形用户界面 (GUI) 来定义, 配置, 分析和观察安全数据。也能通过一种有限命令行的实用程序来做配置和分析。GUI 接口是很容易使用的, 因此推荐使用它。例如, 当执行分析时, 当前的设置在推荐设置模板的下面显示。颜色, 字体, 图标的使用可用来帮助标识存在潜在问题的领域, 且能用鼠标的点击来进行修正。此外, 以前的设置也记录在事件中, 可恢复成原来的设置。

322

有了安全配置编辑器, 基于 Windows 2000 的系统有两种模式的安全分析。第一种模式称为已配置系统分析。这种操作模式是指对由配置模板配置好的, 而不是按事先分析得到的性能来配置的系统进行分析, 分析利用已建立的数据库作为这些模板的结果。第二种安全分析模式称为未配置系统分析, 这种操作模式是将系统与基模板进行比较的分析。基模板用于比较分析的全过程, 且与数据库输出一一起存储, 用于以后的分析。通过改变基模板和安全配置编辑器中的重配置选项, 管理员可更新系统设置。

12.8.2 加密文件系统

加密文件系统 (EFS) 包含在 Windows 2000 操作系统的核心部分。由于分布式系统的部件是连接到网络的, 可能是 Internet。一旦安全泄漏的事件发生, 所有文件都以加密格式存储是非常有益的。Windows 2000 的 EFS 采用了非对称和对称加密。EFS 指定的步骤如下:

1. 用户的明文文件使用随机生成的文件密钥 (FEK) 加密。FEK 采用对称加密。当前, Windows 2000 选择的私钥技术是 DES, 但以后推出的 EFS 版本允许替代现有的加密方法。

2. FEK 密钥作为属性存储在文件中。具体地说: 它是存储在使用用户公钥解密的数据解密区域中, 以及使用恢复密钥的数据区域中, 如图 12-8 所示。如果文件属于多个用户, 则数据解密区域为每个文件所有者配备一个加密 FEK 密钥。

3. 为了让用户解密文件, 采用用户私钥检索 FEK, 随后利用 FEK 来得到文件的内容。数据恢复区域的使用将在后面详细讨论。解密可一块一块地执行, 只有物理地访问专门的块才被解密。

323

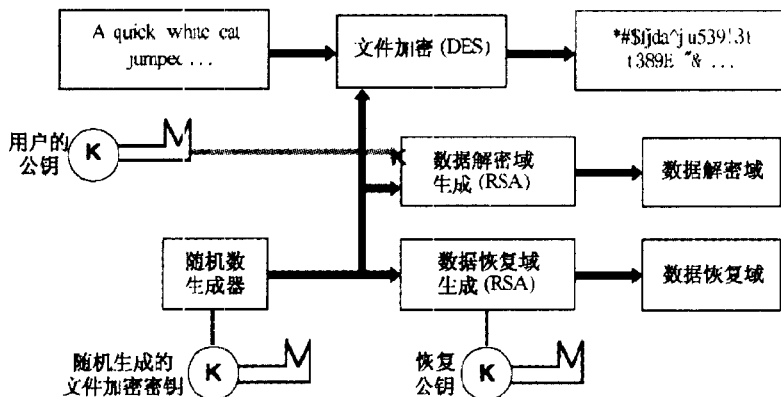


图 12-8 EFS 加密过程

GUI 环境中提供 EFS 能力, 通过 Windows 2000 Explorer 菜单内容以及命令提示符实施。而管理人员不需要介入去启动对文件的加密。如果用户没有公共密钥对, EFS 为用户自动生

成一个。如果目录标记为加密，所有文件和子目录自动地被加密。每个文件有惟一的加密密钥去帮助和保证合适的更名操作。每当一个文件被加密，尽管该文件已更名或移至一个未加加密标记的目录中，它仍将保持加密状态。

如果用户没有某一文件的私钥，则将被拒绝访问该文件。如果没有 Windows 2000 操作系统，整个访问文件机构的机制将是失调的。因此，该方案防止了通过重新启动一个不同的操作系统，从而绕过本系统安全性防护设施而遭受攻击。

EFS 作为一种集成的系统服务来运行，可使它易于管理且难以攻击。EFS 与 NTFS 紧密集成，加密文件的所有临时拷贝也都是加密的，这种能力能扩展到远程文件服务，但 EFS 只对磁盘上的数据进行加密。Windows 2000 操作系统提供的网络协议，与在第 11 章中描述的 SSL/PCT 相比，只能用来对在整个网络中传输的数据进行加密。

在商业环境中，一名雇员可能不通知公司就终止其工作合同，公司就需要一种安全系统能在它的资源中访问该雇员的信息。Windows 2000 操作系统通过包含在数据恢复分区内的内容，用 EFS 提供内在的数据恢复支持。恢复能力应在域管理员的控制之下，但也能指派给指定的数据安全管理员，以提供更好的控制性和灵活性。实际的恢复能力通过恢复密钥提供，它是在每一域的域控制器上定义的。一个给定的系统能配置多个恢复密钥，存储在数据恢复分区中。在恢复过程中，只有系统的私钥需要用来恢复 FEK，因此不会泄漏其他的私有信息（如用户私钥）。

324

12.8.3 微软安全支持提供者接口

微软安全支持提供者接口（SSPI）是一个定义良好的 API，能访问身份认证，消息整合，私密消息以及分布式应用中的安全质量服务。SSPI 可按一种或两种方式使用。第一种方式，SSPI 可通过应用接口直接用于诸如 DCOM（第 6 章）、安全 RPC、WinSock 2.0 和 WinInet 等服务。我们首先研究一个 SSPI 的 API 及其附带软件包的能力。

SSPI 由四个基本的 API 组所组成。第一组 API 是证书管理 API，这一组的接口提供了凭证的访问权，如口令，安全票据等。给出的证书管理 API 如表 12-3 所示。

表 12-3 凭证管理 API

API 名称	说 明
AcquireCredentialsHandle	需要一个指向引用凭证的句柄
FreeCredentialsHandle	此方法释放凭证及其有关资源
QueryCredentialAttributes	此方法允许查询凭证属性，如关联名和域名

第二组 SSPI API 是内容管理 API。内容管理提供了建立和应用各种安全文本的方法，无论是在服务器方或客户方建立。内容管理 API 如表 12-4 所示。

第三组 SSPI API 是消息支持 API。这些 API 提供了通信集成和私有服务，消息支持 API 在表 12.5 中展示。

第四组即最后一组 SSPI API 是包管理 API。这些 API 能访问不同的安全包，各安全包是由指定的安全提供者支持的。包管理 API 如表 12-6 所示。

进一步对 SSPI API 的扩充是为加密技术规划提供消息支持的程序。一般说来，SSPI 允许使用各种安全软件包而不需要改变安全服务接口。通过 SSPI 管理功能，应用程序可列出和

选择一个安全软件包来支持它的需求。安全提供者是一个动态连接库用来实现 SSPI。SSPI 使用一个或多个安全包对应用程序可用,并使该安全包的 SSPI 功能与给定的安全协议的实现相映射,例如 Kerberos 或 SSL。包的名称是在标记该指定包的初始化过程中采用的。

325

表 12-4 内容管理 API

API 名称	说 明
InitializeSecurityContext:	该 API 用于生成一个安全令牌来初始化某一安全内容,该令牌被认为是一种能送到服务器的不透明的消息
AcceptSecurityContext	该 API 利用从客户端收到的不透明消息来创建安全内容
DeleteSecurityContext	该 API 释放安全内容和所有相关联的资源
QueryContextAttributes	该 API 允许查询各种内容属性
ApplyControlToken	该 API 使用一个附加的安全信息到已有的安全内容中
CompleteAuthToken	为 DCE RPC 之类的协议完成一个身份认证的令牌,每当传送更新的某些消息字段时需要修正安全信息
ImpersonateSecurityContext	该 API 将客户端安全内容作为假冒令牌,附加到调用线程
RevertSecurityContext	该 API 停止假冒安全内容和以它的默认初始安全令牌调用线程

表 12-5 消息支持 API

API 名称	说 明
MakeSignature	该 API 生成一个安全签名,该签名基于消息和安全内容
VerifySignature	该 API 验证接收到的签名是否与所需签名匹配

表 12-6 包管理 API

API 名称	说 明
EnumerateSecurityPackages	该 API 列出所有可用的安全软件包及其功能
QuerySecurityPackageInfo	该 API 能查询单个安全包所具有的能力

326

安全包的各种能力实际决定了应用程序利用包能获得怎样的服务。能力可分成三个组:一个能力组是有关身份认证的能力,包括纯客户身份认证、连锁身份认证(微软称为“多腿”认证)、和 Windows 2000 假冒。第二个能力组是传输层(第 1 章中讨论)相关能力,包括报文方式传送、面向连接传送以及数据流连接语义。最后的能力组是关于消息的能力,包含对消息集成和消息私有化的支持。

12.9 HYDRA——一个瘦客户

Hydra 是 Windows 2000 瘦客户的代码名称。该瘦客户为 Windows 2000 服务器操作系统提供基于 Windows 的终端支持,为整个 Windows 操作系统系列提供超级瘦客户。其目的是用一种低成本的终端提供有效的基于 Windows 环境的管理。通过这种终端,用户将能通过 Windows 2000 服务器实施远程访问和商务活动。但实际的终端只用做登录以及与服务服务器通信,并没有完整的功能,如存储文件到该终端可用的本地活动磁盘上,而是严格地限制访问整个网络环境。对于瘦客户只有终端上的本地资源才允许连接和实现服务器功能。对熟悉 UNIX 世界的人来讲,此时瘦客户等效于 X 终端。

12.10 小结

总之，Windows 2000 承诺并设计成能比 NT 4.0 操作系统提供更优良的性能，特别是有关支持分布式计算等方面。几乎基本 NT 技术模式的每个主要方面，都已重新设计成符合当今的应用需求，即要求每个操作系统都能提供分布式操作系统的功能。虽然早期系统积累的经验已提供了足够的知识库，但 Windows 2000 中的设计决策是否是完全分布式级别的真正先进的计算，还将由时间来证明。时间也将告知针对应用需求的这些解决方案是否行之有效。可以确信，将来这种操作系统和其他操作系统将在 Windows 2000 操作系统广泛推广的时候，会在此技术基础上更加完善。无论在哪一种平台上开发，很难相信 Windows 2000 在将来的计算中不会产生重大的影响——特别是分布式计算。

曾经有一种保守的观点，认为计算永远不可能在桌面上实现，Internet 只是为了科学家而设，分布式计算知识只有专家需要，而不是对所有的计算机学家。相信这种观点很快会不攻自破，因为计算机的发展日新月异。

12.11 参考文献

下列白皮书参考文献能提供 Windows 2000 更详细的信息和基础技术的有关协议。微软的 NT 服务器白皮书地址为：www.microsoft.com/ntserver/nts/techdetails [Micr98]。另外在关于 Windows 2000 传统研究期刊中有一些论文，如 [CAPK98, GaShMa98, Micr98 和 Sol98]。

微软 NT 和 Windows 2000 操作系统白皮书系列在 Internet 上提供了一些最好的技术信息，涉及到微软的 Windows 2000 操作系统。关于某些使用协议的信息也能在 Internet 上找到。LDAP 的 RFC 放在下列 URL 中。

- ◆ <http://www.cis.ohio-state.edu/htbin/rfc/rfc2251.html>
- ◆ <http://www.cis.ohio-state.edu/htbin/rfc/rfc2252.html>
- ◆ <http://www.cis.ohio-state.edu/htbin/rfc/rfc2254.html>
- ◆ <http://www.cis.ohio-state.edu/htbin/rfc/rfc2255.html>
- ◆ <http://www.cis.ohio-state.edu/htbin/rfc/rfc2307.html>

LDAP 在线版本用户手册能在 <http://www.umich.edu/~dirsvcs/ldap/doc/man/> 找到。另外，包含 FAQ 站点的与 LDAP 文档有多个链接的页面能在 <http://www.umich.edu/~dirsvcs/ldap/doc/> 找到。

习题

- 12.1 贯穿 Windows 2000 整个设计特别强调的是提供 API，提供实际的、专门的 API 的优越性是什么？
- 12.2 讨论将子系统合并到 Windows 2000 系统结构中的两个优点。
- 12.3 列出为提高 Windows 2000 互操作性和硬件独立性而采用的三个主要的设计要素。说明为什么每个要素能提供这些好处。
- 12.4 修改日志的三个主要优点是什么？为什么？
- 12.5 讨论对等复制策略对大规模的、可扩展的分布式系统的优越性。
- 12.6 如果分布式环境允许从使用 USN 中获益，会有什么不同？

- 12.7 操作系统中并入即插即用功能的三个优点是什么?
- 12.8 利用再解析点的什么技术能非常容易完成任务, 而没有这种技术的话就难以实现。
- 12.9 为什么 Windows 2000 中的索引服务器的服务与老的单用户带软盘驱动器的 DOS 相比较更有效? 列举并描述三个原因。
- 12.10 列出并说明活动目录相对于 X.500 的三个优点?
- 12.11 集群抽象能支持多少种类型的透明性? 为什么所列出的每种类型是集群所固有的?
- 12.12 讨论在分布式操作系统中嵌入 EFS 技术的优点。
- 12.13 讨论你在 Windows 2000 中最喜欢的新特性, 并说明为什么它们对分布式计算是有益的。
- 12.14 为什么 Windows 2000 运行在 64 位地址空间的时候需要改变?

附录 A 外科手术调度程序

外科手术调度程序作为一个实例用来补充本书第一部分中出现的内容，尤其是它演示了 pthread（见第 2 章）、INET 套接字（见第 3 章）和同步互斥（见第 4 章）的应用。该程序在一个 C/S 环境中实现，客户端是一个请求外科配置的外科医生系统，服务器端是用于维护外科配置调度的医院系统。外科医生发出一个请求并将其送到医院端，医院端在有效的时间内做出响应来安排医务调度配置。医生端再选择一个时间并向医院端发回确认请求，而医院端确认或忽略该请求。

A.1—A.5 节是程序的文档部分。A.1 节是文档的概要，A.2 节是有关设计的文档资料，A.3 节是函数描述和函数接口，A.4 节则包含数据字典，A.5 节是用户手册，包括安装说明和执行指令。从 A.6 到 A.8 节包含客户端，服务器端源代码和相应的通用源代码。

331

需要指出的是，该代码的意图是清楚地说明分布式程序的特征，并不是为企业开发的有效程序，因此许多设计要点，比如使用常数表示外科医生的姓名，并不推荐或鼓励在这里使用。之所以这样设计是为了保证代码尽可能简单地揭示分布式特征。

A.1 文档概要

每部分源代码的安排顺序是这样的：首先是主函数文件，然后是该程序的 makefile 文件，再接下去是头文件，最后是源代码文件。每个代码文件在文件头的注释中列出了在该文件中出现的函数。另外还有一个必须但仅在服务器和客户端文档中引用的程序是初始化文件程序。它的操作在 A.5 中描述，而其函数描述在 A.3 中给出，源代码则在 A.8 节中说明。它仅仅是一个用于建立结构文件的简单程序，需要对每个服务器运行一次，并应在安装时执行。

通过对数据流图的分析可知，图中的圆圈表示一个函数，还可知道读者在何时可引用 A.3 节中的函数描述去理解每个函数操作。数据流图见图 A-1—A-11，其中客户端数据流图

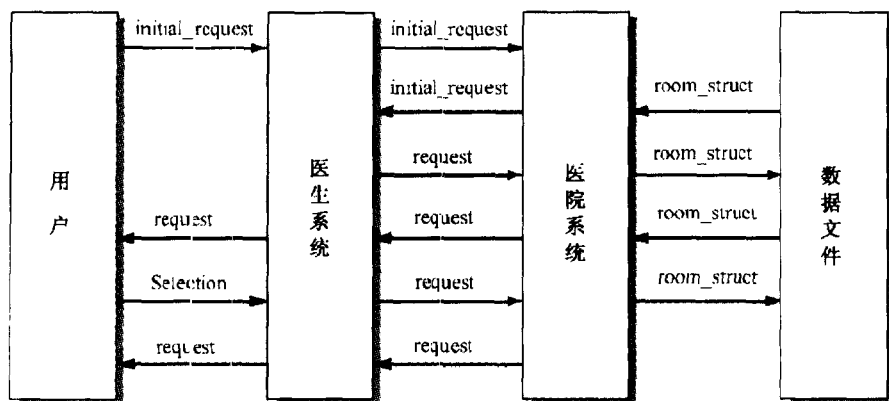


图 A-1 外科手术调度程序流概要图

• in
Selection
request
initial_request

• out
initial_request
request

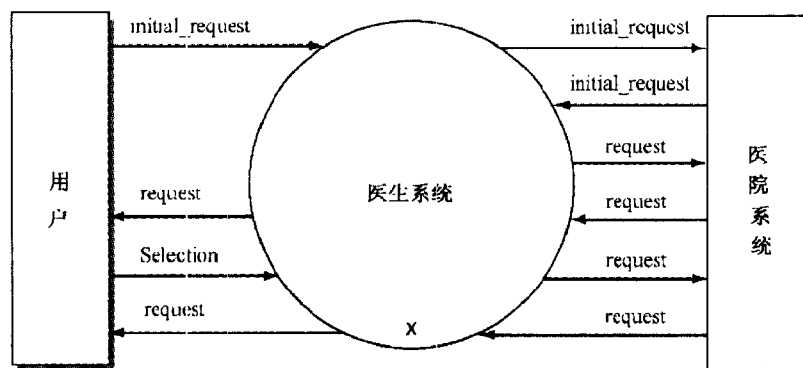


图 A-2 医生数据流概要图

• in
Selection
request
initial_request

• out
initial_request
request

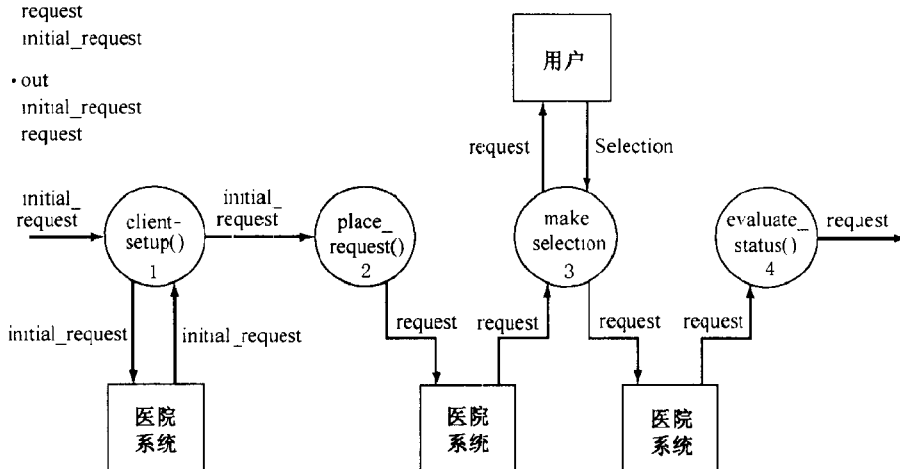


图 A-3 医生系统数据流图

见图 A-2—A-4，服务器端流图见图 A-5—A-11。图 A-1 描述了整个程序数据流的概要。图 A-2 是医生程序数据流概要。图 A-3 是详细的医生系统数据流图，而图 A-4 则是医生进行选择的数据流图。

图 A-5 是医院系统数据流概要图，图 A-6 是详细的医院系统数据流图，图 A-7 是处理初

始请求的数据流图，图 A-8 表示建立/确认可用性的数据流。图 A-9 表示观察文件状态的数据流，图 A-10 表示评价病房选择的数据流图，图 A-11 是关闭服务器的数据流图。

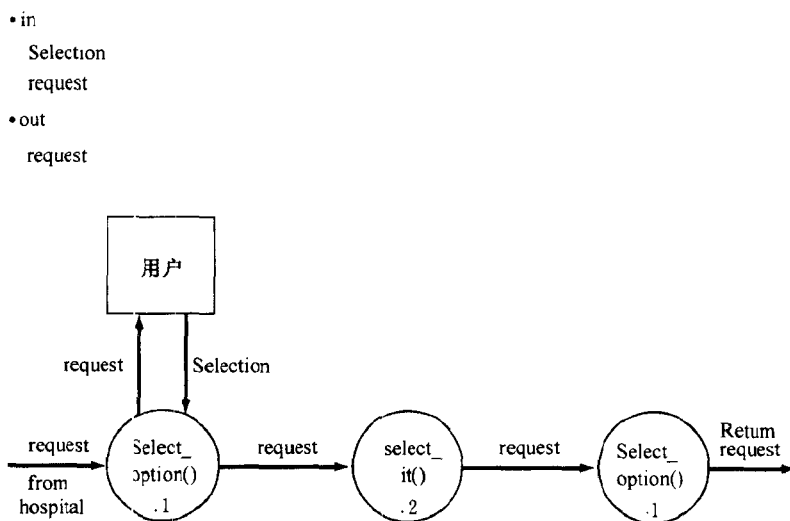


图 A-4 医生进行选择的数据流图

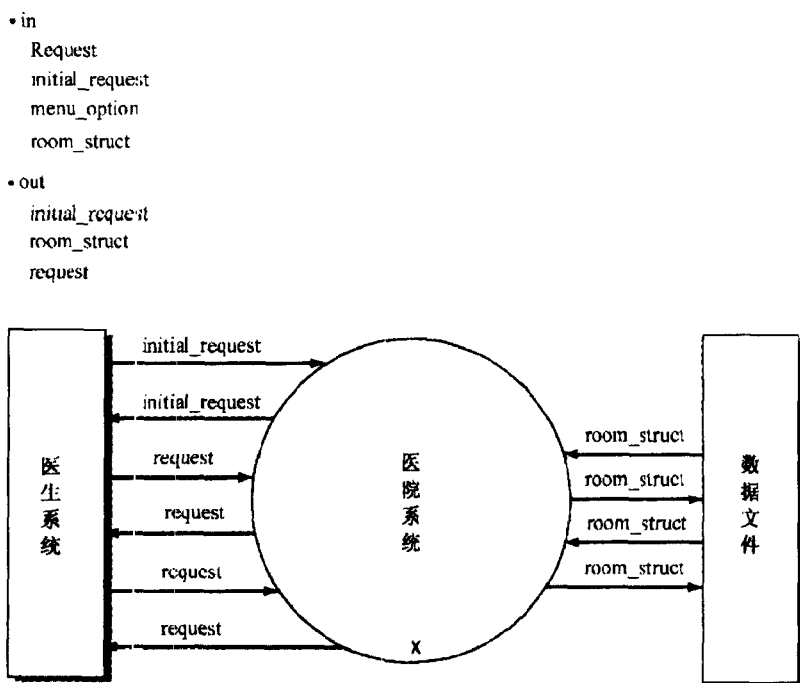


图 A-5 医院数据流概要图

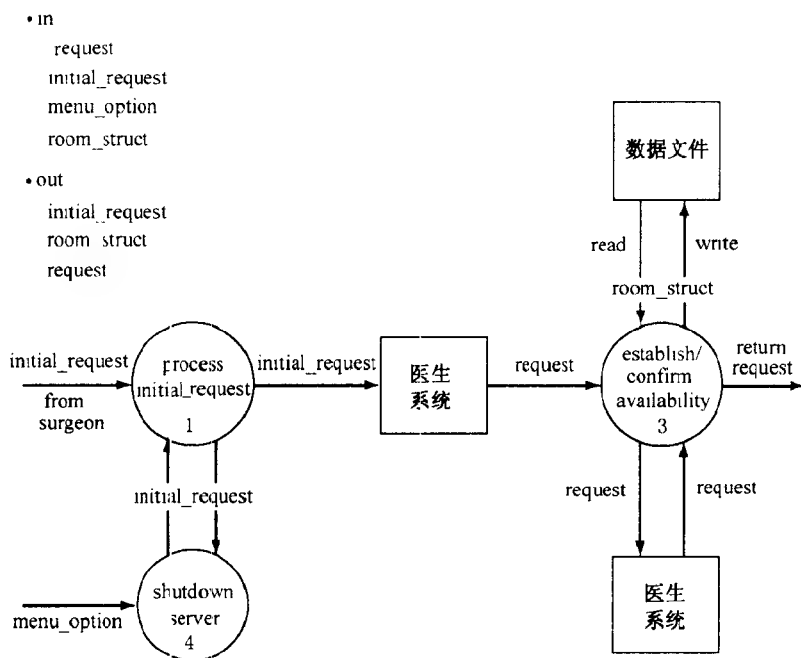


图 A-6 医院系统数据流图

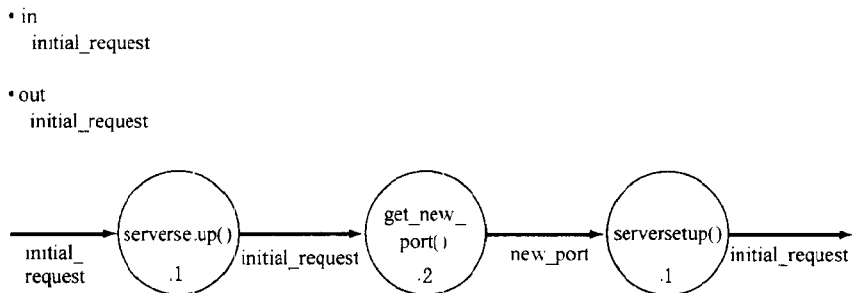


图 A-7 医院系统处理初始化请求数据流图

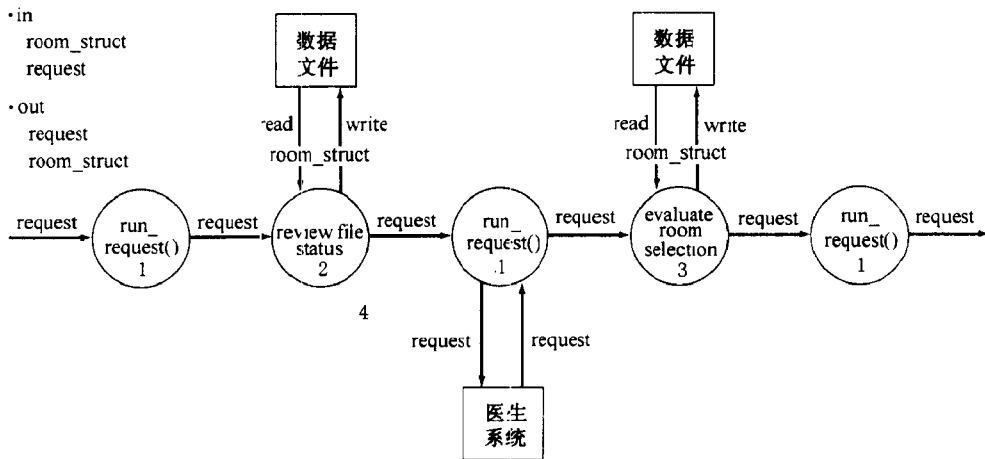
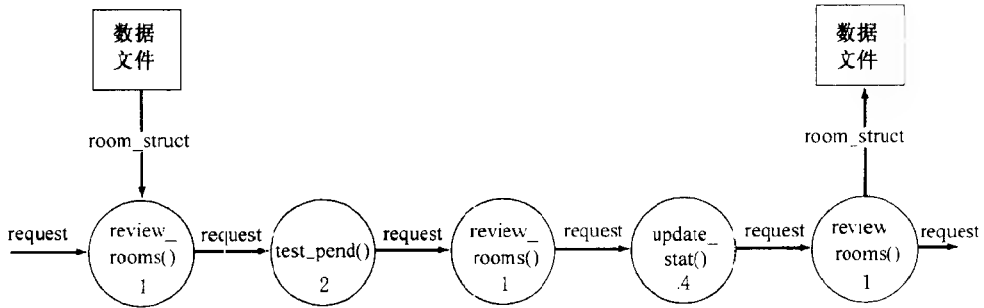


图 A-8 医院系统建立/确认可用性的数据流图

• in
room_struct
request
• out
room_struct
request



336

图 A-9 医院系统观察文件状态的数据流图

• in
room_struct
request
• out
room_struct
request

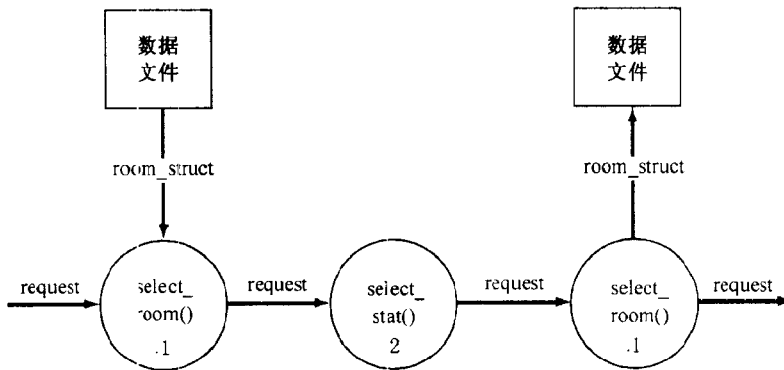


图 A-10 医院系统评价病房选择的数据流图

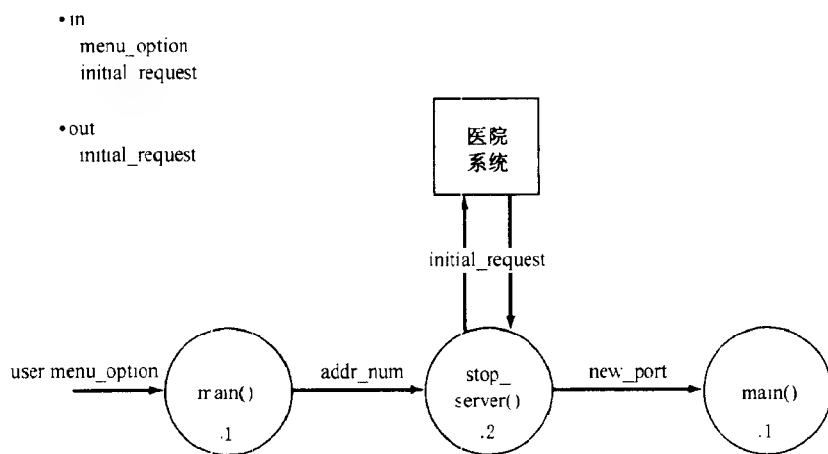


图 A-11 关闭服务器的数据流图

337

A.2 设计文档

外科手术调度程序确切地说是由两个程序组成的，即客户端程序（外科医生）和服务器端程序（医院）。这两个程序用 ANSI C 程序设计语言在 UNIX 环境中实现。套接字用 INET 数据报实现，线程用 POSIX 标准 pthread 库实现，而同步机制则通过 pthread 互斥锁来实现。该程序不打算对年份的跳跃进行修正，也不认为在星期五调度结束之后时间将跳到星期一。我们假设外科医生端每周有七天可进行调度，以小时为单位的时间段从 0800 到 1600。

在设计概要中并没有提到程序中使用的所有函数，那些主要用于简化和改善编程风格的函数没有被提到，因为它们会对程序的总体流程图造成混淆。在 A.3 节函数描述中列举了所有使用过的函数。

在服务器初始运行之前，必须先运行文件初始化程序。由于该程序不是一个用户选项，因此应将它加载到在其标题之下的文档中并且必须和 C/S 程序分开运行。

对客户和服务器程序分别使用它们各自对应的 makefile 文件进行初始化。该 makefile 文件包含一些构造命令，用以说明一些相关性。初始化完成后，服务器程序必须最先开始运行，因为程序交互就是通过客户发出服务请求而开始的。

服务器程序使用命令 hospitalrun 来启动。当输入该命令后，服务器程序从 main() 开始执行，并产生一个线程 main_thread 以接受来自客户的请求。然后服务器的 main() 在一个 while 循环中等待用户输入。用户可以使用有效的选项来关闭服务器。在线程创建过程中先调用函数 serversetup() 来初始化一个套接字端口，然后函数进入一个 while 循环并处于接收状态等待用户请求。main_thread 也将处于接收状态直到有客户发出请求或用户打算关闭服务器。

客户端操作使用命令 surgeonrun 来启动。当输入该命令后，客户端 main() 函数进入 do-while 循环以等待用户输入。用户可以选择发送一个请求也可选择退出关闭该客户程序。如果客户选择选项 R 发送请求，则函数 clientsetup() 被调用。该函数初始化一个客户端套接字端口并发送消息 initial_request 到服务器的已知端口。initial_request 是一个结构体类型 initial_connect 的变量，该结构中的数据域包括枚举类型 dr_id 和整型变量 new_port。每

个特定的用户程序中定义了固定的 `dr_id`，所以用户不需要知道这个信息。

当 `initial_request` 被运行于函数 `serversetup()` 中的线程 `main_thread` 接收时，它被传送到服务器端函数 `get_new_port()`。该函数返回一个服务器端口号，这个端口可提供给用户与服务器进行连接直到服务完成。接下来服务器函数 `serversetup()` 通过 INET 套接字返回 `initial_request` 给用户，其中包含了客户用来与服务器连接以完成服务的端口号。然后 `serversetup()` 产生一个线程 `surgeon_thread` 来处理将由客户发来的请求。在 `surgeon_thread` 线程产生后，服务器函数 `run_request()` 将被调用，它初始化一个套接字并在由 `get_new_port()` 返回的端口号上等待。接下去该处理线程处于接收状态，等待用户的连接。这时，线程 `main_thread` 已经处理完初始请求，通过 `while` 循环它将回到最初的、在已知端口上等待的状态。

338

客户端函数 `clientsetup()` 接收由服务器发回的新端口值并将该端口值返回到主函数 `main()` 中。`main()` 再调用另一个客户端函数 `place_request()` 用上述端口值初始化一个套接字并且与用户连接，因此可以得到客户请求的时间 `julian_date` 和所需要的外科医疗类型。然后 `place_request()` 再将 `request` 传送到服务器。`request` 是一个结构体类型 `day_request` 的变量，它的数据域中包含枚举类型的“`dr_id`”，整型的 `julian_date`，枚举类型的 `surgery_type` 以及范围从 `status0800` 到 `status1600` 的枚举类型 `room_status`。

运行服务器函数 `run_request()` 的线程 `surgeon_thread` 接收到 `request`，然后将指向 `request` 的指针作为参数传递到服务器函数 `review_rooms()`，用以判断哪些病房对于当天所需的医疗类型可用。函数 `review_rooms()` 先对将被打开的文件类型设置互斥锁，再打开正确的文件并建立数组 `init_room[366]`。数组类型是名为 `room_struct` 的结构体类型，它的数据域包含整型的 `julian_date`，范围从 `status0800` 到 `status1600` 的枚举类型 `rooms_status`，从 `ptime0800` 到 `ptime1600` 的长整型值以及从 `dr0800` 到 `dr1600` 的枚举类型 `dr_id`。消息 `request` 根据有关病房的相应文件的当前状态进行更新。可以用三种可能的状态来更新 `request` 状态：即 `busy`、`open` 或 `pending`。如果请求的病房已被提交，则将 `request` 的状态域改为 `busy`，例如 `status0800 = busy`。如果在过去的十分钟里病房状态作为 `open` 已被传递到另一个客户，则 `request` 的状态域用 `pending` 更新。若病房可用，则对该请求时间用 `open` 更新 `request` 的状态域，同时数组用服务器当前时间和发送 `request` 的医生的“`dr_id`”进行更新。若数组的状态域用 `pending` 更新，则表明虽然病房状态为 `open`，但在大概十分钟之前该状态已被发送到另一用户，所以在十分钟过去之前，病房对其他用户不可用。更新结束后，将数组内容写回文件，解除互斥锁。程序将已更新的 `request` 返回到 `run_request()`。`run_request()` 再将 `request` 传回正处于接收状态等待答复的客户。

客户函数 `place_request()` 接收到上述的消息后，调用函数 `select_option()` 并将 `request` 的地址作为参数传递给它。`Select_option()` 给用户提供了一个接口，允许他们选择一个由服务器返回并被标志为 `open` 状态的时间。各个选项显示在屏幕上，用户需要选择其中一个有效的的时间。如用户输入 0000，则表示没有做出任何有效选择并且将退出该程序。选择完毕后，`select_room()` 修改 `request` 的相关内容，控制转回到客户端函数 `place_request()`。`Request` 再次由该函数发送到正在等待客户响应的服务器线程。

线程 `surgeon_thread` 运行 `run_request()` 时接收 `request`。一旦接收到 `request`，服务器函数 `select_room()` 被调用，`request` 的地址作为参数传递给它。同样，`select_room`

() 也对将要打开的文件上锁, 正确打开文件并建立数组 `init_room` [366] (该数据结构前面已说明过)。然后函数遍历 `request` 的数据域中的时间变量以找到由用户 `Select` 的时间, 这个值表示用户需要请求使用的时间段。函数再比较 `request` 中已选时间的 `room_status` 和同一时间数组元素中的 `room_status`。若数组数据域“`dr_id`”和消息 `request` 数据域 `dr_id` 的值相同, 则用 `confirm` 更新 `request` 的 `room_status`, 同时将数组相应的 `room_status` 设置为 `busy`。最后将数组写回文件, 关闭文件并解除互斥锁。控制返回到 `run_request ()`。`run_request ()` 将已被修改的 `request` 送到客户端, 再关闭该套接字退出 `surgeon_thread` 线程。

客户函数 `place_request ()` 通过套接字再次接收到 `request`, 再将 `request` 显示出来供用户核对, 最后还进行状态的评价, 让用户明白他的请求是被确认还是被拒绝。若被拒绝, 函数会列举出所有可能的原因。于是控制返回到客户主函数 `main ()`, 在这里用户可以发送另一个请求或选择结束客户程序。

服务器线程 `main_thread` 仍然处于接收状态监听是否有来自于其他用户的请求。对该线程而言, 用户只有终止服务器这个惟一选择。若用户选择了该选项, `main ()` 用已知的端口号作为参数调用 `stop_server ()`。`stop_server ()` 创建一个套接字并通过 `INET` 连接将“`dr_id`”域为 `master` 的消息 `initial_request` 传送到函数 `serversetup ()`。`serversetup ()` 将这个“`dr_id`”值传递给 `get_new_port ()`, 后者返回值 `99999`。这个值会中断 `serversetup ()` 中的 `while` 循环。然后 `serversetup ()` 将所有文件加锁 (所以没有一个文件能被打开), 再把 `initial_request` 返回到 `stop_server ()` 以告知所有文件已被锁住。`serversetup ()` 最后关闭其套接字并退出 `main_thread` 线程。程序控制返回到 `stop_server ()` 再由 `stop_server ()` 返回到 `main ()`。由于所有文件已被上锁, `main ()` 调用 `pthread_kill_other_threads_np ()` 终止所有剩下的正在运行的线程。至此, 程序将进入一个 `if` 语句, 而用户可选择是否需要程序显示一天中各种类型病房的调度情况。如果用户输入 `R` 来选择显示, 则程序会调用函数 `print_files ()`。

若用户输入的选择不是 `R`, 则程序终止。在输入 `R` 的情况下, `print_files ()` 会进入一个 `do-while` 循环显示用户选择的任何一天的病房调度情况。显示后若用户输入 `Z`, 则控制从 `print_files ()` 返回到 `main ()`。服务器程序将终止并且只能用命令 `hospitalrun` 来重新启动。

A.3 函数功能描述

本节中将要描述函数功能以及函数接口。A.3.1 描述主要的客户端函数, A.3.2 描述主要的服务器端函数。客户端和服务端公用的函数在 A.3.3 节中说明。

A.3.1 客户函数的功能描述

该节主要说明如下几个函数的功能: `clientsetup ()`、`evaluate_status ()`、`main ()`、`place_request ()`、`select_it ()` 和 `select_option ()`。图 A-12 是这些函数的状态图。

```
int clientsetup(int addr_num)
```

该函数用于初始化客户。它用一个已知的服务器端口值 `addr_num` 作为参数, 用这个参

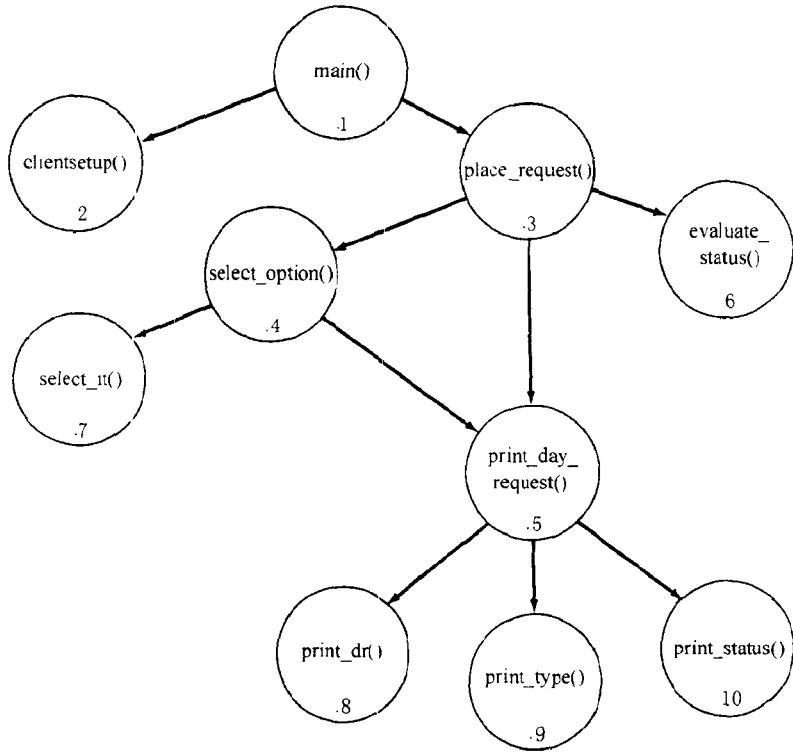


图 A-12 客户/外科医生函数的状态图

数将用户与服务器套接字绑定。函数首先声明一个名为 `initial_connect` 的结构体类型变量 `initial_request`，然后将该用户的枚举型值“`dr_id`”赋给 `initial_request` 的相应数据域。对客户而言，该值是个常数。`initial_request` 的 `new_port` 域被初始化为 0，它可以被其他函数赋予任何整数值。接着函数创建一个 `INET` 数据报类型的套接字将 `initial_request` 用数据报发送到服务器，然后 `clientsetup()` 一直等待直到收到从服务器返回的 `initial_request`，它带回一个整型新端口值 `new_port`。当该值从服务器返回后，`new_port` 返回到调用程序。

```
int evaluate_status(struct day_request *request)
```

该函数的参数是一个指向结构体 `day_request` 的指针变量 `request`，函数评价 `request` 所指的状态值并告诉用户所发生的事。函数体中有一个 `if` 语句用于测试 `request->status0800` 到 `request->status1600` 的值。只要其中有一个的值为 `confirm`，函数就会向用户显示消息 Your request has been confirmed。否则，就会显示出对可能原因的解釋信息。函数返回值 0。

```
void main()
```

客户主函数 `main()` 声明了一个字符变量 `user_select` 和一个整型变量 `new_port`。函数首先显示欢迎界面，然后进入一个 `do-while` 循环以等待用户用 `scanf()` 输入 `user_select`。用户有两个有效的选项，R 表示要发送请求，Q 表示退出程序，其他任何选择都会导致重新进入 `do-while` 循环。在 `do-while` 循环中有一个开关语句，它使用 `user_select` 作为测试表达式。如果选择了 R，函数会用参数 `KNOWN_PORT` 来调用 `clientsetup()`。`KNOWN_PORT` 是

一个在文件 `commonh.h` 中定义的值，它是服务器提供给所有初始请求的公用端口。`clientsetup()` 于是建立一个套接字并向服务器发送数据报，最后由初始请求消息返回一个端口值，返回值存放在变量 `new_port` 中。`clientsetup()` 返回该值后，`place_request()` 被调用并且用 `new_port` 作为参数。接下来的请求消息交给函数 `place_request()` 去处理。`place_request()` 完成其处理后，重新进入 `main()` 的 `do-while` 循环以等待用户发出另一个请求或退出程序。若用户选择了 R，则重复上述序列；若选择了 Q，则客户主函数 `main()` 终止，以后只能用命令 `surgeonrun` 来重新启动它。

```
int place_request(int new_port)
```

本函数使发出请求的客户和一个专用端口上的服务器进行通信。整型参数 `new_port` 表示专用端口的值。函数首先声明一个结构体类型 `day_request` 的变量 `request`，客户用该变量向服务器发送请求，服务器也用它返回一些状态信息。然后再声明一个整型变量 `my_sock` 标记用于连接的套接字。接着声明整型的 `julian_date` 和 `Selection`，其中 `julian_date` 用于读取来自于用户的请求日期，而 `Selection` 用于存储对菜单的选择。然后函数创建一个套接字和服务器绑定。接下去函数进入一个 `do-while` 循环以接收用户输入的请求日期 `julian_date`。这个循环保证用户所提交的日期介于 1 到 365 之间。输入合法日期后，程序再进入另一个 `do-while` 循环等待用户输入表明其所需进行外科手术的类型。同样这个循环保证 `Selection` 值在所定义的界限中。然后 `Selection` 作为一个开关语句的测试表达式，根据用户的选择将正确的枚举型值 `surgery_type` 赋予 `request.surgery_name`。再将 `request.status0800` 到 `request.status1600` 赋值为 `open`。由于服务器的操作依赖于该初始化请求信息，所以必须要对上述变量赋值 `open`。如果该消息没有初始化为 `open` 状态，则服务器不能用有效的时间来更新 `request`。初始化完成后，`request` 以数据报的形式发送到服务器。接下去，`place_request()` 一直等待以接收服务器的响应。当它收到来自服务器的数据报时，`request` 中的时间域已更新为满足客户所需要的手术类型的日期与时间。然后 `place_request()` 调用函数 `select_option()` 并将 `request` 的地址作为参数传递给它。`select_option()` 与用户交互并用用户的选择修改 `request`。`select_option()` 返回后，`place_request()` 用数据报将 `request` 发往服务器，并再次处于等待接收状态。当它再次接收到服务器响应后，用 `request` 作为参数调用 `print_day_request()`，该函数显示出的内容供用户查看。显示完成后，`print_day_request()` 返回调用函数。接着 `evaluate_status()` 被调用，它用一个指向 `request` 的指针作为参数。`evaluate_status()` 显示信息以告诉用户他的请求是否被服务器确认，若被拒绝则显示相关的原因。完成 `evaluate_status()` 的任务后关闭套接字，`place_request()` 返回 0。

```
int select_it (enum room_status *this_room_status,int time_Select)
```

该函数检查 `this_room_status` 的状态并确定用户的选择是否有效。函数用一个 `if` 语句来考察 `this_room_status` 的值。如果 `this_room_status` 的值为 `open`，则将 `this_room_status` 更新为 `select` 状态，这个值表示该选择有效。于是函数返回 0。如果 `this_room_status` 不是处于 `open` 状态，函数立即显示一条信息表明这个选择的时间无效，然后显示出该无效时间 `time_Select` 并返回 1。

```
int Select_option (struct day_request *request)
```

使用该函数用户可以从所有有效的时间中选择一个他所需要的时间来安排外科手术。函数的参数是从服务器（医院系统）返回的当前消息 `request` 的地址。函数中声明了一个整型变量 `time_Select` 用于存储用户输入的选择，还声明了另一个整型变量 `valid_Selection` 把它作为一个标志来确定用户输入的有效选择。如果 `valid_Selection` 的值为 1（为假）则函数进入一个 `while` 循环，直到 `valid_Selection` 的值变成 0（为真）。如果用户选择了一个处于 `open` 状态的时间，说明该时间不是 `busy` 也不是 `pending`，那么循环结束。首先调用函数 `print_day_request()`，它的参数是一个指向消息 `request` 的指针。它向用户显示一个可用时间的列表。`print_day_request()` 函数返回后，用户立即输入他们将要选择的时间 `time_Select`。然后用 `time_Select` 的值作为一个开关语句的测试表达式。如果它的值与 `request->status0800-request->status1600` 中某一个状态为 `open` 的相等，就将 `request->statusXXXX` 更新为 `Select` 状态，并将 `valid_Selection` 的值改为 0 从而退出 `while` 循环。如果 `time_Select` 的值不等于 `request->status0800-request->status1600` 中任何一个状态为 `open` 的，函数就会显示出一条信息表示所选的时间无效，然后重新进入 `while` 循环。如果用户输入一个值 0，则表示不做出任何选择并退出 `while` 循环。当 `valid_Selection` 被设置成 0 时循环终止，函数返回值 0。

A.3.2 服务器函数功能描述

本节描述几个主要的服务器函数：`get_new_port()`、`main()`、`print_files()`、`review_rooms()`、`run_request()`、`select_room()`、`select_stat()`、`server-setup()`、`stop_server()`、`test_pend()` 和 `update_stat()`。图 A-13 是这些函数的状态图。

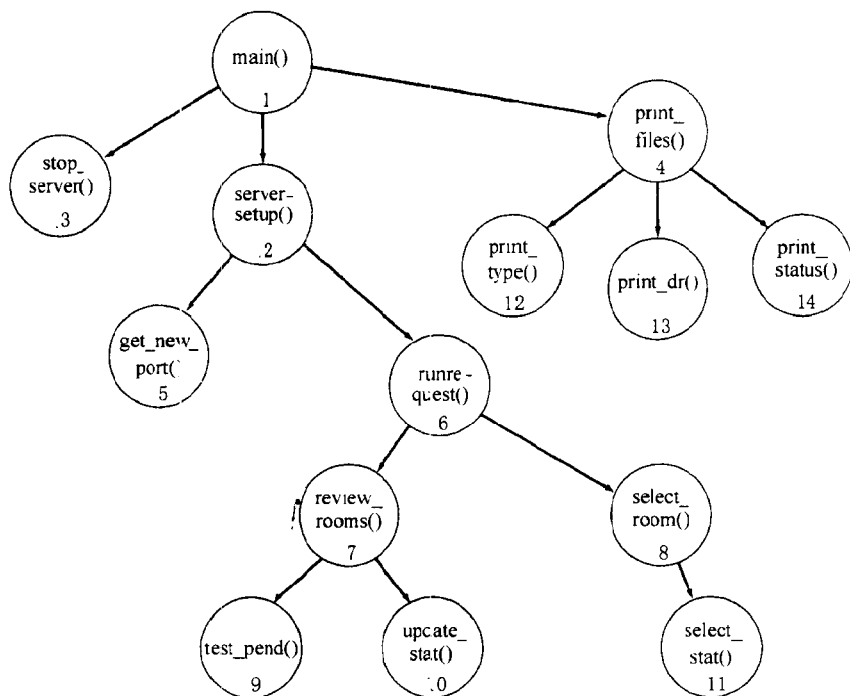


图 A-13 服务器函数状态图

```
int get_new_port(struct initial_connect update_port)
```

该函数用于维护有效“dr_id”列表并且返回服务器上为它们保留的端口号。它用一个结构体类型 `initial_connect` 变量作为参数，参数中包含了枚举类型的“dr_id”。函数用一个开关语句测试该值，用 `dr_id` 作测试表达式。函数最后返回分配给那个“dr_id”的整型端口号。

```
void main()
```

服务器主函数 `main()` 首先声明一个名为 `main_thread` 的 `pthread` 线程，然后再声明一个指向整型值 `addr_num` 的指针并为其分配内存，`addr_num` 用来存放服务器的一个公共的端口地址 `KNOW_PORT`。此外函数还有一个字符变量 `menu_option`。`main_thread` 用函数 `pthread_create()` 创建，在线程创建过程中会调用 `serversetup()`。生成的主线程 `main_thread` 还可以产生其他线程来处理外科医生的请求。这一切做好后，`main()` 进入一个 `do-while` 循环等待用户输入 `menu_option`。此时服务器将阻塞在函数 `scanf()` 处以等待用户输入惟一有效的选择来关闭服务器。在这里我们用字符 `&` 作为惟一有效的输入，因为 `&` 不常用，更容易避免由于误操作而导致的服务器关闭。`do-while` 循环一直等待用户的输入直到用户输入字符 `&` 时才退出。在循环进行的时间里，线程 `main_thread` 正在处理发到服务器的所有初始请求并产生一些子线程处理和客户的通信。当用户输入 `'&'` 时，`main()` 调用 `stop_server()` 并向它传递一个指向 `KNOW_PORT` 的指针变量 `addr_num`。函数 `stop_server()` 将使服务器对所有的临界区（文件）上锁，然后退出线程 `main_thread` 并安全地关闭服务器。一旦 `stop_server()` 结束，`main()` 会调用 `pthread_kill_other_threads_np()` 将终止所有的线程。于是 `main_thread` 及其子线程都将终止。接下去，`main()` 通过使用一个 `if` 语句允许用户显示文件。如果用户选择 R 要求显示，那么调用 `print_files()`，它可以显示任何一天和任何文件状态的信息。如果用户没有选择 `print_files()`，或者选择的显示已结束，则程序终止，以后只能用命令 `hospitalrun` 重新启动服务器。

344

```
int print_files()
```

该函数用于在服务器端显示文件信息。它先声明结构体类型 `room_struct` 的数组 `init room [366]` 用来存放文件信息。函数主体是一个 `do-while` 循环，它一直循环直到用户要求退出。在这个循环体之中还有一个 `do-while` 子循环体，它等待用户输入一个有效的选择。函数会显示出用户对需要打开和显示的文件的选择。如果用户输入有效的选择，函数正确地打开文件并且用文件中的数据初始化数组 `init_room [366]`。接着系统提示用户输入需要显示的日期，这时会进入另一个 `do-while` 子循环以确保用户输入的值介于 1 和 365 之间。当用户进行有效的选择后，程序调用函数 `print_type()`，`print_status()`，`print_dr()` 和 `printf()` 将当天的信息输出给用户，然后文件关闭。用户重新回到主循环的开头再次做出选择。如果用户选择了 Z，主循环结束，函数终止并且返回 0。

```
int review_rooms(struct day_request *request)
```

该函数用来检查医院有哪些病房可提供给客户（医生）安排手术。它的参数是一个指向当前请求变量的指针，该请求变量的所有时间域被初始化为 `open` 状态。函数声明一个字符

数组变量 `roomtype` 和结构体数组 `init_room` [366]，前者用来表示需要用哪一个文件来建立数组，后者用来存放被选文件中的信息。接下来的开关语句用 `request->surgery_name` 作为测试表达式，它确定外科医生所需的病房类型。根据房间的类型，函数 `strcpy()` 将 `roomtype` 赋值为将要使用文件的文件名，再调用 `pthread_mutex_lock()` 为必要的文件操作上锁。然后打开文件并用文件中的数据初始化数组 `init_room` [366]。接下去程序调用函数 `test_pend()` 若干次，这个函数的参数为一对变量的地址，这些值分别为 `init_room[i].status0800—init_room[i].status1600` 和 `init_room[i].ptime0800—init_room[i].ptime1600`。函数 `test_pend()` 用于测试 pending 状态是否结束，并用相关的信息更新变量 `init_room`。再接着 `review_rooms()` 调用 `update_stat()` 若干次，像上面的 `test_pend()` 一样，地址分别为 `init_room[i].status0800—init_room[i].status1600`，`init_room[i].ptime0800—init_room[i].ptime1600`，`request->status0800—request->status1600` 以及 `request->“dr_id”`。`update_stat()` 检查每一个 `room_status` 是否为 open 并相应地更新 `request` 和 `init_room` 的值。完成操作后处于读状态的文件关闭，然后再打开它进行写操作。将 `init_room` 的值写回文件后，程序用 `request->surgery_name` 作为一个开关语句的测试表达式来决定应该解除哪一个互斥锁。在写文件关闭后，打开互斥锁，函数返回 0。

```
void run_request(int *new_addr)
```

该函数在线程 `surgeon_thread` 创建后被调用，它在一个已分配给客户的端口上处理来自于该客户的请求，该端口是为客户所专用以完成和服务器的通信。

函数首先用 `new_addr` 的值创建并绑定一个套接字，接着该套接字处于接收状态等待从客户发往该端口的请求消息。它所接收到的数据报中包含一个结构体类型 `day_request` 的变量 `request`。函数用该变量的地址值调用 `review_rooms()`，`review_rooms()` 用有效的时间值更新消息 `request`，从而可以让用户根据该消息的信息来做出选择。当 `review_rooms()` 完成时，`request` 通过数据报传回用户端。`run_request()` 再次进入接收状态等待客户的响应。当函数再次接收到消息 `request` 时，用户已经用其选择的时间对 `request` 进行了更新。程序用 `request` 的地址值调用函数 `select_room()` 对适当的文件进行更新并进行必要的检查。`select_room()` 完成后，`run_request()` 将确认信息连同已更新的 `request` 一起发回客户端。最后关闭套接字退出线程 `surgeon_thread`。

```
int select_room(struct day_request *request)
```

该函数更新相应的病房文件并确认用户选择病房的日期和时间。它的参数是一个指向结构体类型 `day_request` 的指针，该结构体变量中有一个时间的值已初始化为 Select 状态。函数先声明一个字符数组 `roomtype` 表示需要用哪一个文件来创建数组，然后进入一个用 `request->surgery_name` 作为测试表达式的开关语句来确定用户（医生）选择的病房类型。函数 `strcpy()` 根据房间的类型将 `roomtype` 赋值为即将使用的文件的文件名，`pthread_mutex_lock()` 则对开关语句之后的文件操作加锁。接着打开正确的文件并用文件中相关的值初始化数组 `init_room`。然后函数用多个 if-else 语句来比较 `request->status0800—request->status1600` 中哪一个状态为 Select。若某个时间的状态值为 Select，则函数 `select_stat()` 被调用，并用 `request.statusXXXX` (`XXXX = 0800—1600` 中的一个值)，`init_room`

[i] .statusXXXX, request->dr_name 和 init_room [i] .drXXXX 的地址作为参数。Select_stat () 根据传递给它的参数值来确定或拒绝状态为 Select 的请求, 并对请求进行相应的修改。Select_stat () 返回后, 用于读的文件关闭, 用于写的文件打开并将 init_room 的信息写回文件。接下来程序用 request->surgery_name 作为一个开关语句的测试表达式来决定应该解除哪一个互斥锁。在写文件关闭后, 打开互斥锁, 函数返回值 0。

346

```
int select_stat(enum room_status *request_room_status, enum
room_status *file_room_status, enum "dr_id" *request_dr, enum
"dr_id" *file_dr);
```

该函数根据传递来的参数决定接受还是拒绝来自于客户的请求。它的参数是状态为 select 的指针变量 request_room_status。函数首先用一个 if 语句测试 request_dr 的值。如果 request_dr = file_dr 则 file_room_status 更新为 busy, file_dr 不变, request_room_status 更新为 confirm。如果 request_dr != file_dr, 则表明病房已被其他用户选择, 所以 request_room_status 更新为 busy。注意: 如果客户的 ID 与 file_room_status 中的相等, 则不论状态如何, 将该房间分配给客户是安全的。在该客户或 dr_id 改变前其他用户不能申请它, 而且一旦一个病房在第一次被选择, 它将不会处于 open 状态, 除非"dr_id"变成 pending 状态。函数最后返回 0。

```
void serversetup(int *addr_num)
```

在 pthread_creat () 中调用该函数, 它由服务器程序中的线程 main_thread 运行。调用它时需要向它传递一个指向已知端口号的指针 addr_num 用于与服务器套接字绑定。函数首先声明一个线程 surgeon_thread, 用它产生子线程在特定的端口上处理客户请求消息。然后再声明整型指针变量 *new_addr。接着函数创建和绑定一个套接字, 并且初始化一些在子线程中使用的互斥变量。然后函数用一个 while 循环使程序不停地运行直到用户要求服务器关闭 (关闭操作将在本函数功能描述的后面介绍)。一旦进入 while 循环, 线程就处于接收状态等待用户发送一个包含 initial_connect 结构体类型变量 initial_request 的数据报。收到数据报后, serversetup () 调用 get_new_port () 并向它传递参数 initial_request。get_new_port () 返回一个分配给特定用户的端口号, 再用这个返回值更新 initial_request.new_port。接着向客户发送一个包含该新端口号的数据报, 然后创建 surgeon_thread 并调用 runrequest ()。runrequest () 建立一个套接字去处理剩下的客户请求任务。而 serversetup () 则重新回到循环的开头再次等待其他用户的初始请求 initial_request。先前提到的 surgeon_thread 将继续运行以处理用户的其他特殊请求, 直到 stop_server () 向 serverinit () 发送一个 initial_request."dr_id"值为 master 的数据报。serverinit () 收到该数据报后调用 get_new_port ()。由于"dr_id"的值为 master 所以函数 get_new_port () 返回值 99999, 这个值使循环终止。退出循环后, 为每个互斥锁调用函数 pthread_mutex_lock () 以保证数据的完整性。然后 serverinit () 将一个数据报发回给 stop_server (), 该数据报中 initial_request.new_port 的值设置为 99999。这个消息通知 stop_server () 所有的互斥锁已经锁上。最后关闭套接字并调用 pthread_exit () 退出线程 main_thread。

347

```
int stop_server(int addr_num)
```

该函数用来关闭服务器。它的参数是一个用于与服务器套接字绑定的已知端口号。函数首先声明一个类型为 `initial_connect` 的结构体变量 `initial_request`。当该结构体变量访问完所有的临界区后它的“`dr_id`”数据域设置成 `master`，用该值通知服务器进行关闭操作。而 `initial_request.new_port` 的值初始化为 0，它可被赋以任何整型值。然后函数创建一个 `INET` 数据报类型的套接字并将 `initial_request` 传递到服务器，接着函数处于接收状态，等待任何表明已关闭服务器的返回值。最后关闭套接字并将 99999 返回给调用程序。

```
int test_pend(enum room_status *this_room_status long int *this_time)
```

该函数查看房间状态，如果房间的 `pending` 状态已结束就对其进行更新。函数首先声明类型为 `timeval` 的结构体变量 `current_time` 作为请求消息的时间戳。当天的当前时间可由函数 `gettimeofday()` 得到。接着用一个 `if` 语句测试 `this_room_status` 以确定该值是否为枚举型的状态值 `pending`。如果所申请的房间状态为 `pending`，则函数进入另一个 `if` 语句。第二个 `if` 语句比较 `this_time` 和时间戳的当前时间。如果请求的时间已超过十分钟就将 `this_room_status` 更新为 `open`，因为处于 `pending` 状态的时间段已经用完。如果 `this_room_status` 不是 `pending`，或者处于 `pending` 状态的时间不超过十分钟，则 `this_room_status` 仍然保留其原来的值。最后函数返回 0。

```
int update_stat(enum room_status *request_room_status, enum
room_status *file_room_status, long int *this_time, enum "dr_id"
*request_dr, enum "dr_id" *file_dr);
```

该函数根据数据文件的信息更新所申请房间的状态和 `dr_id`。它首先声明类型为 `timeval` 的结构体变量 `current_time` 作为请求消息的时间戳。当天的当前时间可由函数 `gettimeofday()` 得到。接着函数有一个 `if-else` 语句测试 `file_room_status` 从 08:00 到 16:00 是否为 `open` 状态。如果文件中没有一个时间段可用（处于“`open`”状态），则将 `request_room_status` 赋值为数组中的状态值，`file_room_status` 不变。如果 `file_room_status` 是 `open`，就将它改为 `pending`，`current_time` 的值写到 `this_time` 中并将 `request_room_status` 赋值为 `open`。于是客户被告知该时间段可用。客户（医生）现在有十分钟的时间对该消息做出反应或者可能丢失它。`file_dr` 用选择了该时间段的客户“`dr_id`”进行更新。`if-else` 退出，函数返回 0。

348

A.3.3 公用函数的功能描述

本节描述几个客户端和服务端端的公用函数：`print_day_request()`，`print_dr()`，`print_status()`，`print_type()` 和 `run_request()`。

```
int print_day_request(struct day_request request)
```

该函数用于显示来自客户或服务端的请求信息。它的参数是类型为 `day_request` 的结构体变量 `request`。函数显示的信息包括：医生的姓名、请求的日期、手术的类型以及时间从 08:00 到 16:00 的状态。显示医生的姓名可通过调用函数 `print_dr()` 来实现，只需向它传递参数 `request.dr_name`。手术的类型信息可用参数 `request.surgery_name` 调用函数 `print_type()` 来实现。手术的日期则可直接显示出来，因为日期是一个整型变量。至于

病房不同时间的状态信息可通过调用 `print_status()` 来实现, 向它传递的参数为 `request.status0800 - request.status1600`。函数最后返回 0。

```
int print_dr(enum "dr_id" this_dr)
```

该函数显示医生的姓名。它的参数是一个枚举类型 `dr_id` 的变量。函数的开关语句以 `this_dr` 作为测试表达式, 根据表达式的值显示出相关的医生姓名。最后函数返回 0。

```
int print_status(enum room_status this_room_status)
```

该函数显示病房的状态信息, 它的参数是一个枚举类型 `room_status` 的变量。函数的开关语句以 `this room status` 作为测试表达式, 根据表达式的值显示出相关的状态信息。最后函数返回 0。

```
int print_type(enum surgery_type this_type)
```

该函数显示外科手术类型的名称。它使用一个枚举类型 `surgery_type` 的变量作为参数。函数的开关语句以 `this_type` 作为测试表达式, 根据表达式的值显示出相关的类型信息。最后函数返回 0。

A.3.4 文件初始化

```
void main()
```

文件初始化主函数 `main()` 用来初始化服务器中在外科手术调度程序中使用的文件。对服务器而言, 该函数只需调用一次, 因为所有对文件的更新操作都通过服务器函数来实现。它也可用于初始化一些已损坏且不可挽救的文件。警告: 该函数将覆盖所有以前已被调度的信息。

349

函数先声明结构体类型 `room_struct` 的数组 `init_room` [366], 类型为 `timeval` 的结构体变量 `current_time`, 字符变量 `user_select`, 字符数组 `this_file` [] 和整型变量 `l`。数字 366 用来代表一年的 365 天。如: `Julian day 1 = init_room [1]`, `Julian day 2 = init_room [2]` 等等。然后进入一个 `do-while` 循环 (当用户没有选择退出时), 接着进入该循环中的另一个 `do-while` 循环, 第二个循环用于确保用户作出选择的有效性。函数会显示提示信息要求用户选择需要初始化的文件。用户可以从 A—D 中选择 (有效的文件选项) 或选择 Z 退出程序。程序等待用户用函数 `scanf()` 输入 `user_select`。如果用户选择了 A—D 中的一个, `strcpy()` 将适当的文件名复制到数组 `this_file` 中。再调用函数 `Gettimeofday()` 取得当天的当前时间并将它存入 `current_time` 中。然后程序进入一个 `for` 循环, 循环变量从 1 到 366, 用下面的默认值初始化数组 `init_room`:

- ◆ `Init_room [i].statusXXXX = "open",`
- ◆ `Init_room [i].ptimeXXXX = current_time.tv_sec,`
- ◆ `Init_room [i].drXXXX = "none",`
- ◆ `(XXXX = 0800-1600)`

函数会根据用户的选择将数组内容写回相应的文件, 然后文件关闭, 用户回到外层循环进行进一步的选择。这个过程将一直持续下去直到用户选择了 Z, 这时函数退出而程序终止。

A.4 数据字典

假设：

```
alphanumeric = {A...|Z|a...|z|0...|9}
char = {A...|Z|a...|z}
numeric = {0...|9}

FILE =
    *defined in man pages*

pthread_mutex_t =
    *defined in pthread library man pages*

pthread_t =
    *defined in pthread library man pages*

sockaddr_in =
    *defined in socket library man pages*
```

字典：

```
addr_num = [numeric]
cardio_room_mutex = [pthread_mutex_t]
client = [sockaddr_in]
client_len = [numeric]
current_time = [timeval]
day_request = [dr_name + surgery_name + julian_date +
               status0800 + status0900 + status1000 +
               status1100 + status1200 + status1300 +
               status1400 + status1500 + status1600]

dr0800 = ["dr_id"]
dr0900 = ["dr_id"]
dr1000 = ["dr_id"]
dr1100 = ["dr_id"]
dr1200 = ["dr_id"]
dr1300 = ["dr_id"]
dr1400 = ["dr_id"]
dr1500 = ["dr_id"]
dr1600 = ["dr_id"]

"dr_id" =
    ["johnson" "smith" "williams" "jones" "master" "none"]

dr_name = ["dr_id"]
DRNAME = ["dr_id"]
error_num = [numeric]
file_room_status = [room_status]
i = [numeric]
```

```

initial_connect = [dr_name + new_port]
initial_request = [initial_connect]
init_room[] = [room_struct]
julian_date = [numeric]
KNOWN_PORT = [numeric]
known_sock = [numeric]
main_thread = [pthread_t]
menu_option = [alphanumeric]
my_sock = [numeric]
neuro_room_mutex = [pthread_mutex_t]
new_add = [numeric]
new_addr = [numeric]
new_port = [numeric]
ortho_room_mutex = [pthread_mutex_t]
plastic_room_mutex = [pthread_mutex_t]
ptime0800 = [numeric]
ptime0900 = [numeric]
ptime1000 = [numeric]
ptime1100 = [numeric]
ptime1200 = [numeric]
ptime1300 = [numeric]
ptime1400 = [numeric]
ptime1500 = [numeric]
ptime1600 = [numeric]
request = [day_request]
room_file = [FILE]
room_status = ["busy"|"open"|"pending"|"Select"|"confirm"]
room_struct = [julian_date
                + status0800 + ptime0800 + dr0800
                + status0900 + ptime0900 + dr0900
                + status1000 + ptime1000 + dr1000
                + status1100 + ptime1100 + dr1100
                + status1200 + ptime1200 + dr1200
                + status1300 + ptime1300 + dr1300
                + status1400 + ptime1400 + dr1400
                + status1500 + ptime1500 + dr1500
                + status1600 + ptime1600 + dr1600]
roomtype = [alphanumeric]
Selection = [alphanumeric]
server = [sockaddr_in]
SERVER_HOST = [numeric + "." + numeric + "." + numeric
               + "." + numeric ]
server_len = [numeric]
status0900 = [room_status]
status1000 = [room_status]
status1100 = [room_status]
status1200 = [room_status]
status1300 = [room_status]
status1400 = [room_status]
status1500 = [room_status]

```

352

```

status1600 = [room_status]
stop_request = [initial_connect]
surgeon_thread = [pthread_t]
surgery_name = [surgery_type]
surgery_type = ["neuro"|"cardio"|"plastic"|"ortho"]
surg_type = [surgery_type]
this_dr = ["dr_id"]
this_file[] = [alphanumeric]
this_room_status = [room_status]
this_time = [numeric]
this_type = [surgery_type]
thread_sock = [numeric]
time_Select = [numeric]
update_port = [initial_connect]
user_select = [alphanumeric]
valid_Selection = [(0|1)] };

```

A.5 用户文档

本节包含一些运行该程序必要的文档。其中 A.5.1 节是安装说明，A.5.2 节说明了运行程序的信息。

A.5.1 安装说明

服务器的安装需要在编译前设置两个定义：KNOW_PORT 和 SERVER_HOST。前者是服务器接收初始连接请求的公用端口号，后者是服务器的 IP 地址，这两个定义都在文件 commonh.h 中。服务器也要求运行文件初始化程序，该程序的源代码在文件 write_ca.c 中。要运行该程序，安装者需要键入命令 `cc write_ca.c` 进行编译。编译结束后，键入 `a.out` 来运行。文件初始化程序会自动从该点加载运行，它会为 4 种不同类型的手术病房各建立一个文件并用默认值初始化这些文件。

在安装过程中，服务器程序也需要把那些正确的枚举型值在文件 commonh.h 中反映出来，并且用所有有效的“`cr_id`”值更新各个开关语句。相关的开关语句可以在函数 `get_new_port()` 和 `print_dr()` 中找到。

客户端的安装也需要在编译前设置两个定义：KNOW_PORT 和 DRNAME。前者的意义和服务器端的同名变量相同，后者是分配给该用户的“`dr_id`”。对客户端而言，不需其他任何进一步的操作。

353

A.5.2 如何运行程序

注意：在尝试运行该程序之前要确保安装完成!! 详见前一节的安装信息。

运行该程序仅仅要求所有有效文件在同一个 UNIX 文件目录下，且当前版本的 C 编译器在 UNIX 系统中可用。客户端程序和服务器端程序都运行于 Redhat Linux4.2 和 C 编译器 GNU2.7。服务器所需要的文件是：`commonc.c`、`hospital.c`、`inethosp.c`、`serverinit.c`、`write_ca.c`（程序 `init_files` 用于建立并初始化数据文件）、`commonh.h`、`hospital.h` 和 `makefile`（服务器专用）。客户端所需的文件是：`commonc.c`、`surgeon.c`、`inetsurg.c`、`clientinit.c`、`commonh.h`、`sur-`

geon.h 和 makefile (客户端程序专用)。

如果文件是有效的,则必须要保证包含这些文件的目录是工作目录。用户必须键入“makefile”进行编译,链接和绑定。完成后,如果用户处于服务器端,则键入“hospitalrun”启动服务器;若处于客户端则键入 surgeonrun 启动客户端程序。由此看来,两个程序都是独立引导和菜单驱动的。

注意:当 room_struct 显示在服务器端时,这些输出表示在以前客户发出请求的某一天中处于 pending 状态的任何时间段,并且该时间段没有被选择。当客户面前显示一条请求信息时,它会对在每一个时间上处于 open 状态的请求显示 Open。

A.6 客户端源代码

A.6.1 inetsurg.c

```
/*
inetsurg.c 文件是该调度程序中客户端的主函数 main () 文件。
文件中包括的函数是: main ()。
该程序是 John A.Fritz 在 Dr.D.L.Galli 的指导下完成。
*/

#include "commonh.h"

/* 包含一些服务器和客户公用的头文件和定义 */
#include "surgeon.h"

/* 包含为客户专用的头文件和定义 */

/***** main *****/
void main()
{
    enum "dr_id" this_doc;

    /* 声明该变量用于显示欢迎界面 */
    char user_select[1];

    /* 存储用户的选择 */
    int new_port;

    /* 客户与服务器连接的端口号 */
    this_doc = DRNAME;

    /* 给 this_doc 赋值为已定义 id 的名字 */
    printf("\n\n\n Welcome to the Surgeon Scheduling
        Program Dr. ");
    print_dr(this_doc);

    /* 显示医生的姓名,使界面友好 */

do

    /* 一直允许用户发送请求直到用户选择 Q 退出 */
```

```

{
    printf("Enter 'R' to make a request, 'Q' to quit '\n'");
    scanf("%s", user_select);
    user_select[0] = toupper(user_select[0]);
    switch (user_select[0])
    {
        case 'R':
            new_port = clientsetup(KNOWN_PORT);
            place_request(new_port);
            break;
        case 'Q':
            printf("Shutting down system.\n");
            printf("Please Enter 'surgeonrun'
                to begin again '\n'");
            break;
        default:
            printf("You have chosen an invalid
                option.\n");
            break;
    }
    /* 结束 switch */
    /* 结束 while */
    while(user_select[0] != 'Q');
}
/* 结束 main */

```

355

A.6.2 Client Makefile

调度程序中客户端程序的 makefile 文件。

```

surgeonrun:inetsurg.o clientinit.o surgeon.o commonc.o
    cc -o surgeonrun inetsurg.o clientinit.o surgeon.o commonc.o
inetsurg.o:commonh.h surgeon.h
clientinit.o: commonh.h surgeon.h
surgeon.o: commonn.h surgeon.h
commonc.o: commonn.h hospital.h

```

A.6.3 Surgeon.h

/* surgeon.h 文件用于维护客户端所专用的包含文件和定义，外科手术调度程序。

*/

```
#define DRNAME smith;
```

/* 建立从客户端发送请求的医生的"dr_id"，每一个客户端程序应该为相应的医生设置该数据域 */

```
int clientsetup(int addr_num);
```

/* 初始化客户端的函数。传递的参数是用于与服务器套接字绑定的已知端口值。函数处于等待状态接收新端口号。当它接收到从服务器返回的这个值时，新端口号返回到该调用程序 */

```

int place_request(int new_port);

    /* 初始化专用于某个请求的新端口。传递的参数是一个新端口值，这个
    端口专门为该请求服务直到服务完成 */

int select_option (struct day_request *request);

    /* 允许用户根据有效的时间段信息选择一个手术时间。传递的参数是指
    向结构体变量 day_request 的指针。函数调用后按用户的选择更新请求消
    息 */

int evaluate_status(struct day_request *request);

    /* 检测结构体 day_request 变量的状态以告诉用户所发生的事。传递的
    参数是从服务器（医院）返回后的当前结构体的地址。函数根据结构体
    中的信息向用户显示 request 的状态消息 */

int select_it(enum room_status *this_room_status, int
time_Select);

    /* 函数检查 * this_room_status 的值以确定是否为 open 状态。传递的参
    数是枚举类型变量 room_status。如果 * this_room_status = open 则返回值
    为 0，否则向用户报告并返回 1 */

```

356

A.6.4 Clientinit.c

/* clientinit.c 文件描述了外科手术调度程序中的 clientsetup () 函数

包含的函数是：clientsetup ()。

```

*/
#include "commonh.h"

    /* 包含所有公共头文件的本地文件 */

#include "surgeon.h"

    /* 包含所有为客户端（医生）专用的头文件的本地文件 */

struct initial_connect initial_request;

    /* 请求在特定端口建立连接的结构体传送到服务器 */

/***** clientsetup *****/

int clientsetup(int addr_num)

    /* 初始化客户端的函数。传递的参数是用于与服务器套接字绑定的已知
    端口值。函数处于等待状态接收新端口号。当它接收到从服务器返回
    的这个值时，新端口号返回到该调用程序 */

{
    int known_sock,      /* 标志这个特定的套接字 */
        server_len;

    /* 标志服务器信息的长度 */

    struct sockaddr_in server,

```

357

```

        /* 为服务器产生端口的 internet 版本 */
        client;

        /* 为客户产生端口的 internet 版本 */
        struct msghdr msg;

        /* 声明将在 sendmsg 中使用的结构体 */
        struct iovec iov[1];

        /* 存储消息 sendmsg 中的数据和数据长度信息的结构 */
        initial_request.dr_name = DRNAME;

        /* 将以前为该用户定义的值存放在初始化的 request 中 */
        initial_request.new_port = 0;

        /* 发送请求消息到服务器前将 request 的 new_port 初始化为 0 */
        server.sin_family = AF_INET;

        /* 建立 internet 类型的连接 */
        server.sin_port = htons(addr_num);

        /* 分配一个传递给该函数的端口号 */
        server.sin_addr.s_addr = inet_addr(SERVER_HOST);

        /* 寻找 inet 地址，主机的确切地址在文件 commonh.h 中给出 */
        if ((known_sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0)

        /* 为该连接建立一个句柄。连接范围是用于连接不同机器的 AF_INET 网
           络。SOCK_DGRAM 是进行通信的数据报形式。该例子是非连接的。第
           三个参数是使用协议的，0 表示默认值 */
        {
            perror("Client Socket Error ");
            exit(1);
        } /* 结束 if */

        client.sin_family = AF_INET;

        /* 建立 internet 类型的连接 */
        client.sin_addr.s_addr = htonl(INADDR_ANY);

        /* 地址被设置成 INADDR_ANY，允许进行默认分配 */
        client.sin_port = htons(0);

        /* 默认值 0 允许连接到任何端口 */
        if (bind(known_sock, (struct sockaddr *) &client,
                sizeof(client)) < 0)

        /* 将套接字的值与这个套接字绑定。该套接字的值是 known_sock，&client
           是客户信息、网络名和网络协议。第三个参数是指针的字节数 */
        {
            perror("Client Bind Error");
            exit(1);

```

```

} /* 结束 if */
server_len = sizeof(server);
msg.msg_name = (char *) &server;
msg.msg_namelen = server_len;
msg.msg_iov = iov;
msg.msg_iovlen = 1;
msg.msg_control = NULL;
msg.msg_controllen = 0;

/* 为 sendmsg 对 msg_name 结构和 iov 赋值。msg_name 是可选择的，它表示
接收者的地址（连接协议上不需要该选项）。msg_namelen = 名字的长度。
msg_iov = 输入输出缓冲数组。msg_iovlen = msg_iov 中元素的数量。
msg_control = 辅助数据，设置为 NULL。msg_controllen = 接收消息的结果
值 */

iov[0].iov_base = &initial_request;

/* 即将发送的 initial_request 的地址 */
iov[0].iov_len = sizeof(initial_request);
if (sendmsg(known_sock, &msg, 0) < 0)

/* 发送消息到服务器，需要将新的端口值与 msghdr iovec 中的值重新连接
起来。它也将消息发送给客户。第一个参数是套接字文件描述符，第
二个参数是指向 msghdr 的指针，第三个参数是一个标志 */

{
    perror("Sendmsg");
    exit(1);

} /* 结束 if */
if (recvmsg(known_sock, &msg, 0) < 0)

/* 接收来自于服务器的消息。该消息带回一个端口值用于与服务器重新
连接完成通信。第一个参数是套接字文件描述符，第二个参数是指向
msghdr 的指针，第三个参数是一个标志 */

{
    perror("Client Recvmsg Error");
    close(known_sock);
    exit(1);

} /* 结束 if */
close(known_sock);

/* 关闭套接字，不再为该客户事务服务 */
return(initial_request.new_port);

/* 返回专用于完成客户请求的端口值 */
} /* 结束函数 clientsetup */

```


A.6.5 Surgeon.c

/* surgeon.c 维护一些外科手术调度程序中客户端专用的函数。

函数包括：

```
place_request (),
Select_option (),
evaluate_status (),
select_it () .
```

*/

```
#include "commonh.h"
```

```
/* 包含所有公共头文件的本地文件 */
```

```
#include "surgeon.h"
```

```
/* 包含所有为客户端（医生）专用的头文件的本地文件 */
```

```
struct day_request request;
```

360

```
/* 用于从客户端向服务器传送消息和从服务器返回消息的结构体 */
```

```
/* ***** place_request ***** */
```

```
int place_request(int new_port)
```

```
/* 该函数用来在特定的端口上完成 request 的通信。传递的参数是一个新
   端口值，分配它作为请求信息的专用端口直到通信结束 */
```

```
{
```

```
int my_sock,          /* 表明这个特定的套接字 */
```

```
server_len,
```

```
/* 表明服务器信息的长度 */
```

```
julian_date;
```

```
/* 用于读取来自于用户的请求日期 */
```

```
char Selection[1];
```

```
/* 读取用户对所提供选项的选择 */
```

```
struct sockaddr_in server,
```

```
/* 为服务器产生端口的 internet 版本 */
```

```
client;
```

```
/* 为客户产生端口的 internet 版本 */
```

```
struct msghdr msg;
```

```
/* 用于说明 sendmsg 结构 */
```

```
struct iovec iov[1];
```

```
/* 为 sendmsg 存储数据和数据长度信息的结构 */
```

```
request.dr_name = DRNAME;
```

```
/* 给该请求分配一个在文件 surgeon.h 中定义的 # define 值，这个值对每
```

```

        个客户而言必须设置而且不可更改 */
server.sin_family = AF_INET;

    /* 建立 internet 类型的连接 */
server.sin_port = htons(new_port);

    /* 分配一个传递给该函数的端口号 */
server.sin_addr.s_addr = inet_addr(SERVER_HOST);

    /* 寻找 inet 地址, 主机的确切地址在文件 commonh.h 中由
       # define SERVER_HOST 给出 */
if ((my_sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0)

    /* 为该连接建立一个套接字。连接范围是用于连接不同机器的 AF_INET
       网络。SOCK_DGRAM 是进行通信的数据报形式。该例子是连接的。第
       三个参数是使用的协议, 0 表示默认值 */
{
    perror("Client Socket");
    exit(1);
} /* 结束 if */
client.sin_family = AF_INET;
    /* 见前面对服务器的描述 */
client.sin_addr.s_addr = htonl(INADDR_ANY);
    /* 地址设置为 INADDR_ANY, 允许进行默认分配 */
client.sin_port = htons(0);
    /* 默认值 0 允许连接到任何端口 */
if (bind(my_sock, (struct sockaddr *) &client,
        sizeof(client)) < 0)
    /* 将套接字的值与这个套接字绑定。该套接字的值是 my_sock, &client 是
       客户信息, 网络名和网络协议。第三个参数是指针的字节数 */
{
    perror("Client Bind");
    exit(1);
} /* 结束 if */
server_len = sizeof(server);
    /* 来自于客户的请求信息, 其中包含用户要求服务的日期和手术的类型。
       这些信息存放在即将向服务器发送的结构体中 */
do
{
    /* 支持请求 julian_date 直到输入一个有效的日期 */

    printf("What Julian Date are you requesting surgery for ? \n");
    scanf("%d", &julian_date);
} /* 结束 do-while */
while((julian_date < 1) || (julian_date > 365));
request.julian_date = julian_date;
    /* 将用户所请求的值存放在即将传递到客户端的结构体变量 request 中, 然
       后向用户显示一些选项要求他们输入房间的类型 */

```

```

do
{
    printf("\n\n\n What type of surgery would you like to schedule?
        \n");
    printf("Please make Selection out of the following choices: \n");
    printf(" A = neuro surgery \n");
    printf(" B = cardio-vascular surgery \n");
    printf(" C = plastic surgery \n");
    printf(" D = orthopedic surgery \n");
    scanf("%s", Selection);
    Selection[0] = toupper(Selection[0]);
    switch (Selection[0])
        /*根据用户的选择, 对即将发送到服务器的 request 的数据域 surgery 赋值*/
    {
        case 'A':
            request.surgery_name = neuro;
            break;
        case 'B':
            request.surgery_name = cardio;
            break;
        case 'C':
            request.surgery_name = plastic;
            break;
        case 'D':
            request.surgery_name = orthc;
            break;
        default:
            printf("Invalid Selection \n");
    } /* 结束 switch */
    /*将 request 中的每一个房间状态设置为 open。服务根据这些值做出相应动作。如果某个房间不可用。服务器就改变该状态。因此它不会修改处于 open 状态的房间状态, 如果房间可用, 它的状态值仍然为 open */
} /* 结束 do-while */
/*循环一直进行直到用户作了有效的选择*/
while (('Selection[0] != 'A')
    && (Selection[0] != 'B')
    && (Selection[0] != 'C')
    && (Selection[0] != 'D'));
request.status0800 = open;
request.status0900 = open;
request.status1000 = open;
request.status1100 = open;
request.status1200 = open;
request.status1300 = open;
request.status1400 = open;
request.status1500 = open;
request.status1600 = open;
server_len = sizeof(server);
msg.msg_name = (char *) &server;
msg.msg_namelen = server_len;

```

```

msg.msg_iov = iov;
msg.msg_iovlen = 1;
msg.msg_control = NULL;
msg.msg_controllen = 0;
    /*为 sendmsg 对 msg 结构和 iov 赋值。msg_name 是可选择的，它表示接收者
    的地址（连接协议上不需要该选项）。msg_namelen= 名字的长度。msg_iov
    = 输入输出缓冲数组。msg_iovlen=msg_iov 中元素的数量。msg_control= 辅
    助数据，设置为 NULL。msg_controllen= 接收消息的结果值 */
iov[0].iov_base = &request;
    /* 即将发送的 initial_request 的地址 */
iov[0].iov_len = sizeof(request);
if (sendmsg(my_sock, &msg, 0) < 0)
    /*发送 request 消息到服务器希望得到满足请求日期和请求手术类型的时间列
    表。第一个参数是套接文件的描述符，第二个参数是指向 msghdr 的指针
    第三个参数是一个标志 */
{
    perror("Sendmsg");
    close(my_sock);
    exit(1);
} /* 结束 if */
if (recvmsg(my_sock, &msg, 0) < 0)
    /*接收来自于服务器的消息。该消息带回有效时间的列表以完成请求通信。第
    一个参数是套接字文件描述符，第二个参数是指向 msghdr 的指针，第三个
    参数是一个标志 */
{
    perror ("Client Recvmsg Error");
    close(my_sock);
    exit(1);
} /* 结束 if */
Select_option(&request);
    /*将 request 传递到函数 Select_option 以检查服务器指明的哪些时间是有效的
    并允许用户从这些有效时间中选择一个*/
if (sendmsg(my_sock, &msg, 0) < 0)
    /*将请求发送到服务器，请求中包含用户作出的选择。详见前面的 sendmsg */
{
    perror("Sendmsg");
    close(my_sock);
    exit(1);
} /* 结束 if */
if (recvmsg(my_sock, &msg, 0) < 0)
    /*接收服务器的反应，结构体变量 request 告诉用户请求已确认或者要求用户
    重发 request，无论是哪一种情况程序都将结束 */
{
    perror ("Client Recvmsg Error");
    close(my_sock);
    exit(1);
} /*结束 if */
print_day_request(request);

```

```

        /*向用户显示 request 的信息，告诉他们从服务器（医院）返回时的状态信息*/
        evaluate_status(&request);

        /*检查上述状态，并告诉用户其请求已确认或要求他们重新提交请求*/

        close(my_sock);
        /* 关闭套接字，不再为该客户事务服务 */
        return(0);
    } /* 结束函数的 place_request */
}

/***** Select_option *****/
int Select_option(struct day_request *request)
    /* 程序允许用户从可用的时间中选择他们将要安排手术的时间段。传递的参数
    是由服务器返回的当前结构的地址。最后函数按用户的选择更新该结构
    */
{
    int time_Select,
        /* 用于读取用户从所提供的选项中选择的时间项 */
        valid_Selection;
    /* 用于表示用户何时作出了有效选择 */
    time_Select = 0;
    /* 初始化为一个非有效值的任何值 */
    printf("Following is the status of times
        of your request \n");
    valid_Selection = 1;
    /* 设置一个无效状态来检测有效状态的设置 */
    while (valid_Selection != 0)
        /* 循环一直进行到选择一个处于 open 状态的时间。这表明该时间还没被提交
        或者不处于 pending 状态 */
    {
        print_day_request(*request);
        /* 显示传递给该函数的结构体中的可用时间 */
        printf("Please type in the numeric time
            you would like to Select in the \n");
        printf("following format: XX00, where XX
            is the hour. Example: 0800 \n");
        printf("If you would like to return with
            no option", "selected, please enter 0000 \n");
        scanf("%d", &time_Select);
        /* 获得用户选择，然后进入开关语句去处理该选择 */
        switch (time_Select)

```

```

}

case 800:
    valid_Selection =
        select_it (&request->status0800,
            time_Select);
    /* select_it 将鉴别该选择是有效的 (open 状态) 还是已经被别人所选 (不是
       open 状态)。如果选择是有效的, select_it 返回 0, while 循环将终止; 如果
       选择无效, 则返回值为 1, 用户必须再作出选择 */
    break;
case 900:
    valid_Selection =
        select_it (&request->status0900,
            time_Select);
    break;
case 1000:
    valid_Selection =
        select_it (&request->status1000,
            time_Select);
    break;
case 1100:
    valid_Selection =
        select_it (&request->status1100,
            time_Select);
    break;
case 1200:
    valid_Selection =
        select_it (&request->status1200,
            time_Select);
    break;
case 1300:
    valid_Selection =
        select_it (&request->status1300,
            time_Select);
    break;
case 1400:
    valid_Selection =
        select_it (&request->status1400,
            time_Select);
    break;
case 1500:
    valid_Selection =
        select_it (&request->status1500,
            time_Select);
    break;
case 1600:
    valid_Selection =
        select_it (&request->status1600,

```

```

        time_Select);
    break;
case 0:
    valid_Selection = 0;
    break;

    /* 返回没做出任何选择的结构体 */

default:
    printf('Time Selection %d
           is not valid or is not in open status. \n',
           time_Select);
    printf('Please choose again.\n');

} /* 结束 switch */
} /* 结束 while */
} /* 结束函数 Selection_option */

/***** evaluate_status *****/
int evaluate_status (struct day_request *request)
    /* 函数检测结构体变量 day_request 的状态值并告诉用户所发生的事。传
       递的参数是由服务器返回的当前结构的地址。函数根据结构体中的值
       向用户显示有关请求状态的信息 */

{
    /* 若存在一个状态被确认，则告知用户 */
    if ((request->status0800 == confirm)
        || (request->status0900 == confirm)
        || (request->status1000 == confirm)
        || (request->status1100 == confirm)
        || (request->status1200 == confirm)
        || (request->status1300 == confirm)
        || (request->status1400 == confirm)
        || (request->status1500 == confirm)
        || (request->status1600 == confirm))
    {
        printf("Your request has been confirmed.\n");
    }
    /* 结束 if */
    else
        /* 否则向用户解释为什么没被确认 */

    {
        printf("Your request has been denied. \n
               Possible reasons: \n
               1) Your ten minute response window
                  had expired. \n
               2) You chose to not make a selection. \n
               3) System error.\n");
    }
}

```

```

    } /* 结束 else */
} /* 结束函数 evaluate_status */

/***** select_it *****/
int select_it (enum room_status *this_room_status, int time_Select)
/* 该函数检测 *this_room_status 的值是否为 open 状态。需要传递的参数是
枚举类型 room_status 的变量。如果 *this_room_status 的值为 open, 函数
返回 0; 否则向用户显示相关信息并返回 1 */
{
    if (*this_room_status == open)
    /* 如果状态是 open, 表明它是一个有效选择, 所以用该选择更新状态 */
    {
        *this_room_status = Select;
        return(0);
    }
    /* 结束 if */
    else
    /* 否则它是一个无效选择, 向用户显示提示信息并返回 1 */
    {
        printf("Time Selection %d
            is not valid or is not
            in open status. \n", time_Select);
        printf("Please choose again. \n");
        return(1);
    }
    /* 结束 else */
} /* 结束函数 select_it */

```

369

A.7 服务器源代码

A.7.1 lnethosp.c

/* lnethosp.c 是外科手术调度程序中服务器端主函数文件。

包含的函数: main ()。

该程序是 John A.Fritz 在 Dr.D.L.Galli 的指导下完成。

*/

#include "commonh.h"

/* 服务器和客户公用的 .h 文件和定义 */

#include "hospital.h"

/* 仅为服务器使用的 .h 文件 */

pthread_t main_thread;

/* 主线程, 它产生子线程来处理调度请求 */

/***** main *****/


```

void main ()
{
    int error_num;    /* 用于返回错误信息 */
    int *addr_num;    /* 标注主端口地址 */
    char menu_option [1];
        /* 存放用户对菜单的选择 */
    if((addr_num= (int*) malloc (sizeof (int)) == NULL)
        /* 为主端口值分配空间 */
    {
        perror ("malloc");
        exit (1);
    } /* 结束 if */
    ** addr_num= KNOWN_PORT;
        /* 为主端口号赋值，客户向它发出初始连接请求，然后创建主线程启动服
        务器 */
    error_num=
        pthread_create (&main_thread, NULL, (void
            *) serversetup, addr_num));
    if (error_num!= 0)
        /* 产生线程执行函数 serversetup。第一个参数是一个指向线程 ID 的指针，
        第二个参数是一个指向线程属性的指针，在这里用默认值 NULL。第三
        个参数表明在线程创建后即将调用的函数，第四个参数是一个指向已
        知端口的指针 */
    {
        printf ("pthread create failed error: %d\n", error_num);
        exit (1);
    } /* 结束 if */
    else
    {
        printf ("\n\n\n Server Has Started \n");
    } /* 结束 else */
    do
        /* 显示并接受用户的选择 */
    {
        printf ("Please input '&' to shutdown the server \n
            &= Server Shutdown\n");
        scanf ("%s", menu_option);
        /* 获得用户的选择 */
        switch (menu_option [0])
        {

```

```

    case '&':
        printf ("\n\n\n Beginning Server
                Shutdown \n");
        stop_server (KNOWN_PORT);
        pthread_kill_other_threads_np ();
        printf ("Server Shutdown is
                Complete \n");

        break;
    default:
        printf ("\n\n\n You selected an invalid option :
                %c \n",

                menu_option [0]);

        break;
} /* 结束 switch */
} /* 结束 do-while */
while (menu_option [0] != '&');
    /* 不断循环直到用户选择惟一有效的选项'&'时关闭服务器 */
printf ("Enter 'R' if you would like to print
        reports \n");
printf ("Any other character to quit the
        program \n");
scanf ("%s", menu_option);
menu_option [0] = toupper (menu_option [0]);
if (menu_option [0] == 'R')
{
    printf_files ();
} /* 结束 if */
printf ("Program has exited- Enter 'hospitalrun'
        to begin again \n");
} /* 结束函数 main */

```

371

A.7.2 Makefile for Server

调度外科手术程序中服务器端 makefile 的文件。

```

hospitalrun: inethosp.o serverinit.o hospital.o commonc.o
    cc -o hospitalrun inethosp.o serverinit.o hospital.o commonc.o
    -lpthread
inethosp.o: commonh.h hospital.h
serverinit.o: commonh.h hospital.h
hospital.o: commonh.h hospital.h
commonc.o: commonh.h hospital.h

```

A.7.3 Hospital.h

```

/* 该文件维护一些外科手术调度程序中服务器专用的文件和定义 */
#include <sys/times.h>
pthread_mutex_t neuro_room_mutex,
               cardio_room_mutex,
               plastic_room_mutex,
               ortho_room_mutex;
/* pthread 互斥信息, 每种病房类型有一个互斥量, 它一次只允许一个线程访问其数据文件 */

struct room_struct
/* 用于保存房间的有效状态结构化格式 */
{
int julian_date;
/* 手术的请求日期 */
enum room_status status0800;
/* 特定时间的有效状态 */
long int ptime0800;
/* 房间状态设置为 pending 状态的时间戳, 若它大于十分钟, 则用下一个请求的信息为该有效时间重设此时间戳 */
enum "dr_id" dr0800;
/* 选择房间状态为 busy 和 pending 的客户 id, 服务器需要这个值来跟踪是哪一个用户已经预约了一个房间 */

enum room_status status0900;
long int ptime0900;
enum "dr_id" dr0900;
enum room_status status1000;
long int ptime1000;
enum "dr_id" dr1000;
enum room_status status1100;
long int ptime1100;
enum "dr_id" dr1100;
enum room_status status1200;
long int ptime1200;
enum "dr_id" dr1200;
enum room_status status1300;
long int ptime1300;
enum "dr_id" dr1300;
enum room_status status1400;
long int ptime1400;
enum "dr_id" dr1400;
enum room_status status1500;
long int ptime1500;
enum "dr_id" dr1500;
enum room_status status1600;

```

```
long int ptime1600;
enum "dr_id" dr1600;
};
```

```
void serversetup(int *addr_num);
```

/* 初始化服务器和套接字的函数。传递的参数是服务器上一个公用套接字的端口值。服务器绑定在已知的端口，持续运行以等待客户送来的数据报。收到数据报后，该函数产生一个子线程处理该请求 */

```
int get_new_port (struct initial connect update_port);
```

/* 维护有效客户的列表并返回系统中为他们保留的端口号。传递的参数是一个结构体变量，它包含了枚举类型的用户"dr_id"。函数最后返回分配给该用户的端口号 */

```
void run_request (int *new_add);
```

/* 处理进一步的客户请求。传递的参数是一个特定的端口号，该端口号专门提供给客户请求以和服务器套接字绑定。结构体消息 day_request 在客户和服务器之间来回传递。可用的房间信息传递到客户端，客户选择并更新文件，服务器的确认消息返回到客户端 */

```
int review_rooms (struct day_request *request);
```

/* 确定在客户要求安排手术的当天有哪些病房可用。传递的参数是指向当前请求的指针，该请求中的所有将被检查的时间的状态都是 open。函数将对当前请求中那些不可用的时间状态进行更新，可用的时间仍然处于 open 状态 */

```
int select_room (struct day_request *request);
```

/* 更新客户请求房间的当天时间的 room_file 文件。传递的参数是指向结构体变量 day_request 的指针，该变量中有用户的选择。函数将证实该请求中所选的时间是否被认可，并且用保留的时间更新 day_request */

```
int stop_server (int addr_num);
```

/* 终止服务器的函数。传递的参数是已知的端口号。消息用 "master" 作为 "dr_id" 来告诉服务器它已经访问了所有的临界区，要求关闭服务器。所有的文件被关闭。最后函数处于接收状态等待表明服务器已经关闭的返回值 */

```
int print_files ();
```

/* 函数用来在服务器端显示文件。调用函数时建立需要的文件，它会向用户显示一些提示信息询问他们需要显示哪一天的哪一种房间信息。然后将相关的信息显示在屏幕上 */

```
int test_pend (enum room_status *this_room_status,
               long int *this_time);
```

/* 检查传递参数的房间状态，如果 pending 状态已经结束，则对该消息更

375

新。传递的参数是两个指针变量，第一个是指向枚举类型 room status 的指针，第二个是指向房间处于那个状态的时间的指针。函数检查该请求是否已超过了十分钟。如果是，则表明使该时间状态为 pending 的请求已超过了十分钟，用状态值“open”更新文件信息。如果状态不是 pending 或 pending 状态的时间小于十分钟，则不做任何操作 */

```
int update_stat (enum room_status *request_room_status,
                enum room_status *file_room_status,
                long int *this_time,
                enum "dr_id" *request_dr;
                enum "dr_id" *file_dr);
/* 根据数据文件的信息更新被请求房间的状态和"dr_id"。传递的参数是指
   针变量 request_room_status 和"dr_id"，以及文件的时间戳和 file_room_status。函数根据 file_room_status 的值更新这些值 */

int select_stat (enum room_status *request_room_status,
                enum room_status *file_room_status,
                enum "dr_id" *request_dr;
                enum "dr_id" *file_dr);
/* 如果发送的消息中房间状态为 select，则函数更新这个状态值。传递的
   参数是指向 room_status 的指针变量，该变量中状态值为 select。函数根据
   下面的信息更新参数：如果该时间的请求值已被选择，则检查数组
   的状态；如果该 dr_id 是该时间的最后一个 ID，则将此时间更新为 busy
   状态 */
```

A.7.4 Serverinit.c

376

```
/* serverinit.c 初始化服务器的主端口。包含的函数：serverinit () */
#include "common.h"
/* 包含所有公共头文件和定义的本地文件 */
#include "hospital.h"
/* 包含所有为服务器端（医院）专用的头文件的本地文件 */
pthread_t surgeon_thread;
/* 声明线程 id，该线程在专用端口上建立连接 */
struct initial_connect initial_request;
/* 发送到服务器并请求在其专用端口上建立连接的结构体消息 */
/***** serversetup *****/
void serversetup (int *addr_num)
/* 函数用一个已知端口值与服务器套接字绑定。它首先等待客户的请求
   以建立一个专用套接字。然后产生子线程处理请求并将专用端口号返回
   给请求服务的客户，最后它再次回到等待状态接收下一个请求 */
```

```

{
int known_sock,    /* 标志特定的套接字 */
    server_len,
                    /* 表示服务器和客户消息的长度 */
    client_len,
    error_num;
    /* 标志 pthread 创建过程中的错误 */
int *new_addr;
    /* 用于在 pthread_creat 中传递新端口值 */
struct sockaddr_in server,
    /* 为服务器产生端口的 internet 版本 */
    both_server, /*
        client;
        /* 为客户产生端口的 internet 版本 */
struct msghdr msg;
    /* 在 sendmsg 中将要使用的结构体 */
struct iovec iov [1];
    /* 为 sendmsg 存储数据和数据长度信息的结构 */
if ( (known_sock = socket (AF_INET, SOCK_DGRAM, 0)) < 0)
    /* 为该连接建立一个套接字。连接范围是用于连接不同机器的 AF_INET 网
        络。SOCK_DGRAM 是进行通信的数据报形式。该例子是非连接的。第三
        个参数是使用的协议, 0 表示默认值 */
{
    perror ("Server Socket");
    exit (1);
} /* 结束 if */
server.sin_family = AF_INET;
    /* 建立 internet 类型的连接 */
server.sin_port = htons (*addr_num);
    /* 分配传递给函数的端口值 */
server.sin_addr.s_addr = inet_addr (SERVER_HOST);
    /* 寻找 inet 地址, 主机的确切地址在文件 commonh.h 中给出 */
if (bind (known_sock, (struct sockaddr*) &server,
    sizeof (server)) < 0)
    /* 将套接字的值与这个套接字绑定。该套接字的值是 known_sock, &server
        是服务器信息, 网络名和网络协议。第三个参数是指针的字节数 */
{
    perror ("Server Bind");
    exit (1);
}

```

```

    } /* 结束 if */
    pthread_mutex_init (&neuro_room_mutex, NULL);
    pthread_mutex_init (&cardio_room_mutex, NULL);
    pthread_mutex_init (&plastic_room_mutex, NULL);
    pthread_mutex_init (&ortho_room_mutex, NULL);
    /* 为每种房间类型设置并初始化互斥锁。它将保证所有服务器创建的线程
       会对请求线程进行初始化 */
    msg.msg_name = (char *) &client;
    msg.msg_namelen = sizeof (client);
    msg.msg_iov = iov;
    msg.msg_iovlen = 1;
    msg.msg_control = NULL;
    msg.msg_controllen = 0;
    /* 为 sendmsg 对 msg 结构和 iov 赋值。msg_name 是可选择的，它表示接收者
       的地址（连接协议上不需要该选项）。msg_namelen = 名字的长度，msg_
       iov = 输入输出缓冲数组。msg_iovlen = msg_iov 中元素的数量，msg_control
       = 辅助数据，设置为 NULL。msg_controllen = 接收消息的结果值 */
    iov[0].iov_base = &initial_request;
    /* 即将发送的 initial_request 的地址 */
    iov[0].iov_len = sizeof (initial_request);
    if (recvmsg (known_sock, &msg, 0) < 0)
    /* 接收来自于客户端的消息。第一个参数是套接字文件描述符，第二个
       参数是指向 msg_hdr 的指针，第三个参数是一个标志 */
    {
        perror ("Receive Message");
        exit (1);
    } /* 结束 if */
    initial_request.new_port = get_new_port (initial_request);
    /* 给 initial_request.new_port 分配一个与 "dr_id" 相关的值，用于告诉用户他
       们可在那个端口上和服务器线程连接完成通信。在进入下面的循环前必须
       完成这些操作，接下来进入 while 循环 */
    while (initial_request.new_port != 99999)
    /* 持续接收来自于客户的请求直到服务器用 "master" 作为医生的 ID 发送
       一条消息 (get_new_port 返回 99999)。出现这种情况时，一旦所有互斥锁
       在这个线程控制之下则该线程退出 */
    {
        iov[0].iov_base = &initial_request;
        /* 将被发送的 initial_request 的地址赋值 */
        iov[0].iov_len = sizeof (initial_request);

```

```

if (sendmsg (known_sock, &msg, 0) < 0)
    /* 发送消息到客户端, 指明新端口号在 msghdr iovec 中重新进行连接。第
       一个参数是套接字文件描述符, 第二个参数是指向 msghdr 的指针, 第三
       个参数是一个标志 */
{
    perror ("Sendmsg");
    exit (1);
} /* 结束 if */
if ((new_addr = (int*) malloc (sizeof (int))) == NULL)
    /* 为 new_port 分配内存 */
{
    perror ("malloc");
    exit (1);
} /* 结束 if */
*new_addr = initial_request. new_port;
error_num =
    (pthread_create (&surgon_thread, NULL, (void
        *) run_request, new_addr));
    if (error_num != 0)
        /* 产生线程执行函数 runrequest, 该线程处理客户进一步的请求。第一个
           参数是一个指向线程 ID 的指针, 第二个参数是一个指向线程属性的
           指针, 在这里用默认值 "NULL"。第三个参数表明在线程创建后即将
           调用的函数, 第四个参数是一个指向已知端口的指针 */
{
    printf ("pthread create failed error: %d\n",
        error_num);
    exit (1);
} /* 结束 if */
if (recvmsg (known_sock, &msg, 0) < 0)
    /* 接收从客户端发来的消息。第一个参数是套接字文件描述符, 第二个参
       数是指向 msghdr 的指针, 第三个参数是一个标志 */
{
    perror ("Receive Message");
    exit (1);
} /* 结束 if */
initial_request. new_port = get_new_port (initial_request);
    /* 用分配给该客户的端口号更新 initial_request 的端口值 */
} /* 结束 while */
    /* 如果 while 循环退出则表明已发送了一个数据报来关闭服务器。函数保

```


证在服务器关闭之前所有的临界区已上锁。若临界区上锁，则只要有文件处于打开状态，服务器就不能关闭 */

```
pthread_mutex_lock (&neuro_room_mutex);
pthread_mutex_lock (&cardio_room_mutex);
pthread_mutex_lock (&plastic_room_mutex);
pthread_mutex_lock (&ortho_room_mutex);
iov [0] .iov_base = &initial_request;
    /* 为即将发送的 initial_request 的地址赋值 */
iov [0] .iov_len = sizeof (initial_request);
if (sendmsg (known_sock, &msg, 0) < 0)
{
    perror ("Sendmsg Exit of server init");
    exit (1);
} /* 结束 if */
    /* 向客户发回答复以说明所有临界区已经上锁。第一个参数表示套接字的
       文件描述符，第二个参数是指向即将发送消息的指针，第三个参数
       是一个标志，设置为 0 */
close (known_sock);
    /* 关闭套接字不再接收任何客户的请求 */
pthread_exit ((void *) 0);
    /* 用户要求退出线程 */
} /* 结束函数 serversetup */
```

A.7.5 Hospital.c

```
381 /* "hospital.c" 维护调度程序中服务器端专用的函数。包含的函数有：get_new_port ()、
run_request ()、review_room ()、select_room ()、stop_server ()、print_files ()、test_
pend ()、update_stat ()、select_stat () */
#include "commonh.h"
    /* 包含所有公共头文件的本地文件 */
#include "hospital.h"
    /* 包含所有为服务器端（医院）专用的头文件的本地文件 */

/***** get_new_port *****/
int get_new_port (struct initial_connect update_port)
    /* 维护有效客户的列表并返回系统中为他们保留的端口号。传递的参数
       是一个结构体变量，它包含了枚举类型的用户 dr_id。函数最后返回分
       配给该用户的端口号。

```

注意：这仅仅是一个简化的例子。实际中由于端口数有限，通常选择端口的算法很复杂，而且也需要维护一个有效的和可用的端口数的列表以利于选择 */

```

{
    int new_port;
    switch (update_port.dr_name)
    {
        /* 当 Williams 的 new_port = 20001 时, 断开 */
        case jones:
            new_port = 20002;
            break;
        case smith:
            new_port = 20003;
            break;
        case johnson:
            new_port = 20004;
            break;
        case master:
            new_prot = 99999;
            break;
        default:
            new_port = 30000;
            break;
    } /* 结束 switch */
    return (new_port);
}; /* 结束 get_new_port */

/***** run_request *****/
void run_request (int *new_addr)
    /* 处理进一步的客户请求。传递的参数是一个特定的端口号, 该端口号专门
       提供给客户请求以和服务器套接字绑定。结构体消息 day_request 在客户和
       服务器之间来回传递。可用的房间信息传递到客户端, 客户选择及更新后
       写回文件, 服务器的确认消息返回到客户端 */
{
    int thread_sock,
        /* 标志特定的套接字 */
        server_len,
            /* 表示服务器和客户消息的长度 */
        client_len;
    struct day_request request;
        /* 在客户和服务器之间传递消息的结构 */
    struct sockaddr_in server,
        /* 为服务器产生端口的 internet 版本 */

```

383

```

    client;
    /* 为客户产生端口的 internet 版本 */
struct msghdr msg;
    /* 在 sendmsg 中将要使用的结构体 */
struct iovec ioc [1];
    /* 为 sendmsg 存储数据和数据长度信息的结构 */
if ((thread_sock = socket (AF_INET, SOCK_DGRAM, 0)) < 0)
    /* 为该连接建立一个套接字。连接范围是用于连接不同机器的 AF_INET 网
    络。SOCK_DGRAM 是进行通信的数据报形式。该例子是非连接的。第三个
    参数是使用的协议，0 表示默认值 */
{
    perror ("Server Socket Error");
    exit (1);
} /* 结束 if */
server.sin_family = AF_INET;
    /* 建立 internet 类型的连接 */
server.sin_port = htons (*new_addr); /* fix (*new_port) */
    /* 分配传递给函数的端口值 */
server.sin_addr.s_addr = inet_addr (SERVER_HOST);
    /* 寻找 inet 地址，主机的确切地址在文件 commonh.h 中给出 */
if (bind (thread_sock, (struct sockaddr*) &server,
    sizeof (server)) < 0)
    /* 将套接字的值与这个套接字绑定。该套接字的值是 thread_sock，&server 是
    服务器信息，网络名和网络协议。第三个参数是指针的字节数 */
{
    perror ("Server Bind Error");
    exit (1);
} /* 结束 if */
client_len = sizeof (client);
msg.msg_name = (char *) &client;
msg.msg_namelen = sizeof (client);
msg.msg_iov = ioc;
msg.msg_iovlen = 1;
msg.msg_control = NULL;
msg.msg_controllen = 0;
    /* 为 sendmsg 对 msg 结构和 ioc 赋值。msg_name 是可选择的，它表示接收者的
    地址（连接协议上不需要该选项）。msg_namelen = 名字的长度，msg_iov =
    输入输出缓冲数组，msg_iovlen = msg_iov 中元素的数量，msg_control = 辅助
    数据，设置为 NULL。msg_controllen = 接收消息的结果值 */

```

384

```

iov [0] .iov_base = &request;
    /* 即将发送的 initial_request 的地址 */
ivo [0] .iov_len = sizeof (request);
if (recvmsg (thread_sock, &msg, 0) < 0)
    /* 接收来自于客户端的消息。第一个参数是套接字文件描述符，第二个参
       数是指向 msghdr 的指针，第三个参数是一个标志 */
{
    perror ("Receive Message");
    exit (1);
}
review_rooms (&request);
    /* 按可用的房间信息更新当前 request */
if (sendmsg (thread_sock, &msg, 0) < 0)
    /* 发送消息到客户端，指明有效时间。第一个参数是套接字文件描述符，第
       二个参数是指向 msghdr 的指针，第三个参数是一个标志 */
{
    perror ("Sendmsg");
    close (thread_sock);
    exit (1);
}
if (recvmsg (thread_sock, &msg, 0) < 0)
    /* 接收从客户端发来的消息。第一个参数是套接字文件描述符，第二个参数
       是指向 msghdr 的指针，第三个参数是一个标志 */
{
    perror ("Receive Message");
    exit (1);
}
select_room (&request);
    /* 将发送进行有效性检查的消息并更新相应的文件，如果用户的选择被确
       认，则用确认状态更新该消息 */
if (sendmsg (thread_sock, &msg, 0) < 0)
    /* 向客户发回答复表明所有临界区已经上锁。第一个参数表示套接字的文件
       描述符，第二个参数是指向 msghdr 的指针，第三个参数是一个标志 */
{
    perror ("Sendmsg");
    close (thread_sock);
    exit (1);
}
close(thread_sock);
    /* 关闭与该套接字文件描述符相关的连接 */

```

```

pthread_exit(new_addr);
    /* 退出线程并调用相关的处理 */
} /* 结束函数 run_request */

/***** review_rooms *****/
int review_rooms (struct day_request *request)
    /* 确定在客户要求安排手术的当天有哪些病房可用。传递的参数是指向当前
    请求的指针，该请求中的所有将被检查的时间的状态都是 open。函数将对
    当前请求中那些不可用的时间状态进行更新，可用的时间仍然处于 open 状态 */
{
    int i;          /* 用于遍历数组 */
    struct timeval current_time;
        /* 用于表明当前时间和核对 pending 的时间是否大于 10 分钟 */
    char roomtype [25];
        /* 表明该数组中的字符串为当前数组所读的文件类型 */
    FILE *room_file;
    struct room_struct init_room [366];
        /* 从文件中读数据建立该结构，并用来根据需要进行读操作和更新操作 */
    switch (request-> surgery_name)
        /* 判别用户需要的房间类型，然后打开与该类型有关的文件并对该打开
        的文件设置互斥锁。 */
    {
        case neuro:
            strcpy(roomtype, "neuro_rooms.dat");
            pthread_mutex_lock(&neuro_room_mutex);
            break;
        case cardio:
            strcpy(roomtype, "cardio_rooms.dat");
            pthread_mutex_lock(&cardio_room_mutex);
            break;
        case plastic:
            strcpy(roomtype, "plastic_rooms.dat");
            pthread_mutex_lock(&plastic_room_mutex);
            break;
        case ortho:
            strcpy(roomtype, "ortho_rooms.dat");
            pthread_mutex_lock(&ortho_room_mutex);
            break;
        default:
            printf("invalid surgery name \n");
            exit(1);
    } /* 结束 switch */
        /* 打开文件并建立结构体类型 room_struct 数组 */

```

```

if((room_file = fopen (roomtype, "r")) == (FILE *) NULL)
{
    printf ("%s didn't open: %d\n", roomtype, errno);
} /* 结束 if */
for (i = 1; 365 >= i; i++)
    /* 从 room_struct 列表文件中建立数组 */
    {
        fread(&init_room[i], sizeof(init_room[i]), 1, room_file);
    } /* 结束 for */
i = request->julian_date;
    /* 将 i 设置成用户选择的日期 */
gettimeofday (&current_time, NULL);
    /* 取得当天的当前时间让用户判断是否 pending 状态已超过了十分钟 */
test_pend (&init_room[i].status0800, &init_room[i].ptime0800);
    /* 将数值传给函数 test_pend 并判断所需房间的数值是 pending 或更新值。
    函数将检查并判断 request 是否超过 10 分钟。如果是，则用 open 更新数
    组；如果不是 pending，或 pending 状态少于 10 分钟，则以数组的当前
    值对它进行更新。在返回后，客户可判断哪些时间是可用的 */

test_pend(&init_room[i].status0900, &init_room[i].ptime0900);
test_pend(&init_room[i].status1000, &init_room[i].ptime1000);
test_pend(&init_room[i].status1100, &init_room[i].ptime1100);
test_pend(&init_room[i].status1200, &init_room[i].ptime1200);
test_pend(&init_room[i].status1300, &init_room[i].ptime1300);
test_pend(&init_room[i].status1400, &init_room[i].ptime1400);
test_pend(&init_room[i].status1500, &init_room[i].ptime1500);
test_pend(&init_room[i].status1600, &init_room[i].ptime1600);
update_stat(&request->status0800,
            &init_room[i].status0800,
            &init_room[i].ptime0800,
            &request->dr_name,
            &init_room[i].dr0800);

    /* 调用函数 update_stat 判断文件的状态。如果文件中该时间不可用（不处
    于 open 状态），则函数用文件状态更新 request，这表明用户对该房间
    的选择无效。若该时间状态为 open 并且 request 中相关信息以前没有从
    open 状态改变，则将数组写为 pending，这表明用户所需求的该时间可
    用。然后客户有 10 分钟时间对 request 做出响应或丢弃它。再用持有该
    时间段的医生名对文件更新。该名字仅仅允许那个医生访问 10 分钟 */

update_stat(&request->status0900,
            &init_room[i].status0900,
            &init_room[i].ptime0900,
            &request->dr_name,
            &init_room[i].dr0900);

```

387

388

```

update_stat(&request->status1000,
            &init_room[i].status1000,
            &init_room[i].ptime1000,
            &request->dr_name,
            &init_room[i].dr1000);
update_stat(&request->status1100,
            &init_room[i].status1100,
            &init_room[i].ptime1100,
            &request->dr_name,
            &init_room[i].dr1100);
update_stat(&request->status1200,
            &init_room[i].status1200,
            &init_room[i].ptime1200,
            &request->dr_name,
            &init_room[i].dr1200);
update_stat(&request->status1300,
            &init_room[i].status1300,
            &init_room[i].ptime1300,
            &request->dr_name,
            &init_room[i].dr1300);
update_stat(&request->status1400,
            &init_room[i].status1400,
            &init_room[i].ptime1400,
            &request->dr_name,
            &init_room[i].dr1400);
update_stat(&request->status1500,
            &init_room[i].status1500,
            &init_room[i].ptime1500,
            &request->dr_name,
            &init_room[i].dr1500);
update_stat(&request->status1600,
            &init_room[i].status1600,
            &init_room[i].ptime1600,
            &request->dr_name,
            &init_room[i].dr1600);
fclose(room_file);

```

389 /* 关闭读状态的文件，重新打开文件进行写操作来更新数组 */

```

if((room_file = fopen(roomtype, "w")) == (FILE *) NULL)
{
    printf("%s didn't open: %d \n", roomtype, errno);
} /* 结束 if */
for (i = 1; 365 >= i; i++)
    /* 将更新的数组写回文件 */
{
    fwrite(&init_room[i], sizeof(init_room[i]), 1, room_file);
} /* 结束 for */
fclose(room_file);

```

```

switch(request->surgery_name)
    /* 判断需要哪个文件。解除该文件上的互斥锁允许其它线程访问 */
{
    case neuro:
        pthread_mutex_unlock(&neuro_room_mutex);
        break;
    case cardio:
        pthread_mutex_unlock(&cardio_room_mutex);
        break;
    case plastic:
        pthread_mutex_unlock(&plastic_room_mutex);
        break;
    case ortho:
        pthread_mutex_unlock(&ortho_room_mutex);
        break;
    default:
        printf("invalid surgery name \n");
        exit(1);

} /* 结束 switch */
return (0);
} /* 结束函数 review_rooms */

/***** select_room *****/
int select_room (struct day_request *request)
    /* 更新客户房间请求当天时间的 room_file 文件。传递的参数是指向结构体变量 day_request 的指针，该变量中有用户的选择。函数将证实该请求中所选的时间是否被认可，并且用保留的时间更新 day_request。也会用 confirmed 更新 request 以告诉用户他的请求已被接受 */
{
    int i;                /* 用来遍历索引 */
    char roomtype [25];
        /* 该数组中的字符串表明当前数组所读的文件类型 */
    FILE *room_file;
    struct room_struct init_room [366];
        /* 从文件中读数据建立该结构，并根据需要进行读操作和更新操作 */
    switch (request->surgery_name)
        /* 判别用户需要的房间类型，然后打开与该类型有关的文件并对该打开的文件设置互斥锁 */
    {
        case neuro:
            strcpy(roomtype, "neuro_rooms.dat");
            pthread_mutex_lock(&neuro_room_mutex);
            break;
        case cardio:

```



```

        strcpy(roomtype, "cardio_rooms.dat");
        pthread_mutex_lock(&cardio_room_mutex);
    break;
case plastic:
        strcpy(roomtype, "plastic_rooms.dat");
        pthread_mutex_lock(&plastic_room_mutex);
    break;
case ortho:
        strcpy(roomtype, "ortho_rooms.dat");
        pthread_mutex_lock(&ortho_room_mutex);
    break;
default:
        printf("invalid surgery name \n" ;
        exit(1);

} /* 结束 switch */

if ((room_file = fopen (roomtype, "r")) == (FILE *) NULL)
{
    printf("%s didn't open: %d \n",
        roomtype,errno);

} /* 结束 if */

for (i = 1; 365 >= i; i++)
    /* 从 room_struct 列表文件中建立数组 */
{
    fread(&init_room[i], sizeof(init_room[i]), i, room_file);
} /* 结束 for */

i = request->julian_date;
    /* 将 i 设置成用户选择的日期 */

if (request->status0800 == Select)
    /* 如果 request 中该时间值已被选择则检查数组的状态。假如该"dr_id"对
    这个时间而言是最后一个 ID，就将该时间更新为 busy 状态。注意：不
    论何种状态，只要该"dr_id"在这个时间段中就可以安全地分配房间。如
    果客户或姓名改变了，则没有人可以请求房间。也就是说，一旦一个
    房间被选择，则它将不再处于 open 状态，除非在 pending 状态时"dr_id"
    发生变化。另外，由于使用了 else-if 语句，所以只可接受每个请求中
    一个选择 */

{
    select_stat(&request->status0800,
        &init_room[i].status0800,
        &request->dr_name,
        &init_room[i].dr0800);

} /* 结束 if */

```

```
else if (request->status0900 == Select)
{
    select_stat(&request->status0900,
                &init_room[i].status0900,
                &request->dr_name,
                &init_room[i].dr0900);

    /* 结束 else if */
else if (request->status1000 == Select)
{
    select_stat(&request->status1000,
                &init_room[i].status1000,
                &request->dr_name,
                &init_room[i].dr1000);

    /* 结束 else if */
else if (request->status1100 == Select)
{
    select_stat(&request->status1100,
                &init_room[i].status1100,
                &request->dr_name,
                &init_room[i].dr1100);

    /* 结束 else if */
else if (request->status1200 == Select)
{
    select_stat(&request->status1200,
                &init_room[i].status1200,
                &request->dr_name,
                &init_room[i].dr1200);

    /* 结束 else if */
else if (request->status1300 == Select)
{
    select_stat(&request->status1300,
                &init_room[i].status1300,
                &request->dr_name,
                &init_room[i].dr1300);

    /* 结束 else if */
else if (request->status1400 == Select)
{
    select_stat(&request->status1400,
                &init_room[i].status1400,
                &request->dr_name,
                &init_room[i].dr1400);

    /* 结束 else if */
else if (request->status1500 == Select)
{
```

```

select_stat(&request->status1500,
            &init_room[i].status1500,
            &request->dr_name,
            &init_room[i].dr1500);

/* 结束 else if */
else if (request->status1600 == Select)
{
    select_stat(&request->status1600,
                &init_room[i].status1600,
                &request->dr_name,
                &init_room[i].dr1600);

/* 结束 else if */
fclose (room_file);

/* 关闭读状态的文件，重新打开文件进行写操作来更新数组 */

if ((room_file = fopen (roomtype, "w")) == (FILE *) NULL)
{
    printf(" %s didn't open: %d \n",
           roomtype,errno);

/* 结束 if */
for (i = 1; 365 >= i; i++)
    /* 将被更新的数组写回文件 */
    {
        fwrite(&init_room[i], sizeof(init_room[i]), 1, room_file);
    }
/* 结束 for */
fclose (room_file);
switch (request-> surgery_name)
    /* 判断需要哪个文件。解除该文件上的互斥锁从而允许其他线程访问 */
    {
        case neuro:
            pthread_mutex_unlock(&neuro_room_mutex);
            break;
        case cardio:
            pthread_mutex_unlock(&cardio_room_mutex);
            break;
        case plastic:
            pthread_mutex_unlock(&plastic_room_mutex);
            break;
        case crtho:
            pthread_mutex_unlock(&ortho_room_mutex);
            break;
        default:
            printf("invalid surgery name \n");
            exit(1);
    }
}

```

```

    } /* 结束 switch */
    return (0);
} /* 结束函数 select_room */

/***** stop_server *****/
int stop_server (int addr_num)
    /* 终止服务器的函数。传递的参数是与服务器套接字绑定的已知端口号。
       消息用 master 作为 dr_name 来告诉服务器它已经访问了所有的临界区，
       从而要求关闭服务器。所有的文件被关闭。最后函数处于接收状态等待
       表明服务器已经关闭的返回值 */
{
    struct initial_connect stop_request;
        /* 发送到服务器请求关闭服务器的结构体消息 */
    int known_sock,
        /* 标志特定的套接字 */
        server_len;
        /* 表示服务器消息的长度 */
    struct sockaddr_in server, client;
        /* 为服务器和客户产生端口的 internet 版本 */
    struct msghdr msg;
        /* 在 sendmsg 中将要使用的结构体 */
    struct iovec iov [1];
        /* 为 sendmsg 存储数据和数据长度信息的结构 */
    stop_request.dr_name = master;
        /* 用该值表示已经访问了所有的临界区并通知服务器进行关闭操作。在此
           时已关闭所有的文件 */
    stop_request.new_port = 0;
        /* 在向服务器发送消息前将 request 的 new_port 初始化为 0，可根据情况给它
           赋予其他的值 */
    server.sin_family = AF_INET;
        /* 建立 internet 类型的连接 */
    server.sin_port = htons (addr_num);
        /* 分配传递给函数的端口值 */
    server.sin_addr.s_addr = inet_addr (SERVER_HOST);
        /* 寻找 inet 地址，主机的确切地址在文件 commonh.h 中给出 */
    if ((known_sock = socket (AF_INET, SOCK_DGRAM, 0)) < 0)
        /* 为该连接建立一个套接字。连接范围是用于连接不同机器的 AF_INET 网
           络。SOCK_DGRAM 是进行通信的数据报形式。该例子是非连接的。第三
           个参数使用的是协议，0 表示默认值 */
    {

```

394

395

```

        perror ("Server Socket Error");
        exit (1);
    } /* 结束 if */
    client.sin_family = AF_INET;
        /* 建立 internet 类型的连接 */
    client.sin_addr.s_addr = inet_addr (SERVER_HOST);
        /* 将地址设置成 SERVER_HOST, 同一台机器用做客户机和服务器 */
    client.sin_port = htons (0);
        /* 默认值 0 表示可连接到任何端口 */
    if (bind (known_sock, (struct sockaddr *)&client,
        sizeof (client)) < 0)
        /* 将套接字的值与这个套接字绑定。该套接字的值是 known_sock, &client 是
        服务器信息, 网络名和网络协议。第三个参数是地址的字节数。注意: 此
        时的这个调用函数中服务器充当了客户机的角色, 而在 main_thread 中充当
        了服务器的角色 */
    {
        perror ("Server Bind Error");
        exit (1);
    } /* 结束 if */

    msg.msg_name = (char *) &server;
    msg.msg_namelen = sizeof (server);
    msg.msg_iov = iov;
    msg.msg_iovlen = 1;
    msg.msg_control = NULL;
    msg.msg_controllen = 0;

        /* 为 sendmsg 对 msg 结构和 iov 赋值。msg_name 是可选择的, 它表示接收者
        的地址 (连接协议上不需要该选项)。msg_namelen = 名字的长度, msg_iov
        = 输入输出缓冲数组, msg_iovlen = msg_iov 中元素的数量, msg_control =
        辅助数据, 设置为 NULL, msg_controllen = 接收消息的结果值 */
    iov [0].iov_base = &stop_request;
        /* 给即将发送的 initial_request 的地址赋值 */
    iov [0].iov_len = sizeof (stop_request);
    if (sendmsg (known_sock, &msg, 0) < 0)
        /* 向服务器发送消息, 用 master 请求关闭服务器, 传递结构体变量表示返
        回的确认消息。第一个参数是套接字文件描述符, 第二个参数是指向 ms-
        ghdr 的指针, 第三个参数是标志 */
    {
        perror ("Sendmsg in stop_server");
        exit (1);
    }
}

```

```

if (recvmsg (known_sock, &msg, 0) < 0)
    /* 接收服务器的消息。该消息中的端口值是 99999, 表示服务器已经关闭而
       且所有临界区已上锁。第一个参数是套接字文件描述符, 第二个参数是
       指向 msghdr 的指针, 第三个参数是标志 */

{
    peeror ("Client Recvmsg Error");
    close (known_sock);
    exit (1);
} /* 结束 if */
close (known_sock);
    /* 关闭套接字, 不再为该客户事务服务 */
return (stop_request.new_port);
    /* 返回值 99999 作为标志, 表明 master "dr_id" 将进行终止服务器操作 */
} /* 结束函数 stop_server */

/***** print_files *****/
int print_files ()
    /* 函数用来在服务器端显示文件。调用函数时建立需要的文件。它会提供
       一些提示消息以询问用户需要显示哪一天的哪一种房间信息, 然后将相
       关的信息显示在屏幕上 */
{
    enum surgery_type surg_type;
        /* 稍后将为用户显示文件的类型 */
    char user_select [1],
        null_value [1];
        /* 接受用户的输入 */
    int i;
        /* 用来遍历数组 */
    FILE *room_file;
    char this_file [25];
        /* 用来读取文件的名字 */
    struct room_struct init_room [366];
        /* 存储房间有效状态的结构体数组 */
    do
        /* 持续循环直到用户要求退出 */
    {
    do
        /* 持续循环直到用户作出有效选择 */
    {

        printf("Which file would you like to print
            from? \n");
        printf("Please enter the appropriate character
            : \n");
        printf("

```

```

        A = Neurclogical Rooms \n
        B = Cardiovascular Rooms \n
        C = Orthopedic Rooms \n
        D = Plastics Rooms \n
        Z = EXIT \n");
scanf("%s",user_select);
user_select[0] = toupper(user_select[0]);
switch(user_select[0])

    /* 根据用户选择对 surg_type 赋值并复制所需打开的文件名 */

{
    case 'A': strcpy(this_file,"neuro_rooms.dat");
                surg_type = neuro;
                break;
    case 'B': strcpy(this_file,"cardio_rooms.dat");
                surg_type = cardio;
                break;
    case 'C': strcpy(this_file,"ortho_rooms.dat");
                surg_type = ortho;
                break;
    case 'D': strcpy(this_file,"plastic_rooms.dat");
                surg_type = plastic;
                break;
    case 'Z': printf("Exiting Print Program \n");
                break;
    default: printf("This Option is Invalid \n");
                break;

} /* 结束 switch */
} /* 结束 do-while */
while ((user_select [0] != 'A')
        /* 持续循环直到作出有效选择 */
        &&(user_select [0] != 'B')
        &&(user_select [0] != 'C')
        &&(user_select [0] != 'D')
        &&(user_select [0] != 'Z'));
        /* 打开所需的文件建立数组以允许用户对日期和显示数据作出选择 */
if ( user_select[0] != 'Z')
{
    if ((room_file = fopen (this_file, "r")) == (FILE *) NULL)
    {
        printf("%s didn't open: %d \n ", this_file,errno);

    } /* 结束 if */
    for (i = 1; 365 >= i ; i++)
    {
        fread(&init_room[i], sizeof(init_room[i]), 1, room_file);

```

```

    /* 结束 for */

do
{
    printf("What day would you like to look at? \n");
    scanf("%d", &i);

} /* 结束 do-while */
while ( (i < 1) || (i > 365));

    /* 获得用户需要显示的日期。While 循环保证所选的日期在 1 到 365 之间。i
       = 1 表示 Julian 年的第一天 */

    /* 调用相关的函数显示所需的数据 */

printf("Surgery date: %d ", init_room[i].julian_date)
printf(" Room Type ");
print_type(surg_type);

printf("Room Status: \n ");
printf(" 08:00 - ");
print_status(init_room[i].status0800);
printf("Dr. ");
print_dr(init_room[i].dr0800);
printf(" 09:00 - ");
print_status(init_room[i].status0900);
printf("Dr. ");
print_dr(init_room[i].dr0900);
printf(" 10:00 - ");
print_status(init_room[i].status1000);
printf("Dr. ");
print_dr(init_room[i].dr1000);
printf(" 11:00 - ");
print_status(init_room[i].status1100);
printf("Dr. ");
print_dr(init_room[i].dr1100);
printf(" 12:00 - ");
print_status(init_room[i].status1200);
printf("Dr. ");
print_dr(init_room[i].dr1200);
printf(" 13:00 - ");
print_status(init_room[i].status1300);
printf("Dr. ");
print_dr(init_room[i].dr1300);
printf(" 14:00 - ");
print_status(init_room[i].status1400);
printf("Dr. ");
print_dr(init_room[i].dr1400);
printf(" 15:00 - ");
print_status(init_room[i].status1500);
printf("Dr. ");
print_dr(init_room[i].dr1500);
printf(" 16:00 - ");

```



```

print_status(init_room[i].status1600);
printf("Dr. ");
print_dr(init_rocm[i].dr1600);
printf(" \n This is the status of the room you requested.");
printf("ENTER any character to continue \n");
scanf("%s",null_value);
printf("\n\n\n");
fclose(room_file);

```

400 /* 在显示完 room_struct 后，重新回到 do-while 循环中，允许用户选择另一天和另一种类型或选择退出 */

```

    /* 结束 if */
/* 结束 do-while */

```

```

while (user_select [0] != 'Z');
return (0);

```

/* 结束函数 print_files */

/* ***** test_pend ***** */

```

int test_pend (enum room_status *this_room status,
               long int *this_time)

```

/* 检查传递参数的房间状态，如果 pending 状态已经结束，则对该消息更新。传递的参数是两个指针变量，第一个是指向枚举类型 room_status 的指针，第二个是指向房间处于那个状态的时间的指针。函数检查该请求是否已超过了 10 分钟。如果是，则表明使该时间的状态为 pending 的请求已超过了 10 分钟，用状态值 open 更新文件信息；如果状态不是 pending 或 pending 状态的时间小于 10 分钟，则不做任何操作 */

```

{
    struct timeval current_time;
    /* 用于表明当前时间并核对是否 pending 状态的时间超过 10 分钟 */
    gettimeofday (&current_time, NULL);
    /* 取得当天的当前时间让用户判断是否 pending 状态已超过了 10 分钟 */
    if (*this_room_status == pending)

```

/* 如果被请求的房间状态为 pending，则判别时间请求是否已超过 10 分钟。若是，则表明使该时间段为 pending 的请求已超时，则将文件更新为 open。如果状态不是 pending，或 pending 的状态不超过 10 分钟，则用当前数组中的信息更新 request。在该消息返回客户端后，客户可判断哪些时间是可用的 */

```

{
    if ((current_time.tv_sec - *this_time) > 600)

```

/* 如果将房间设置成 pending 状态已超过 10 分钟，则将状态更新为 open 并将时间戳改为当前时间 */

```

        *this_time = current_time.tv_sec;
        *this_room_status = open;
    } /* 结束 if */
} /* 结束 if */
return (0);
} /* 结束函数 test_pend */

/***** update_stat *****/
int update_stat (enum room_status *request_room_status,
                 enum room_status *file_room_status,
                 long int *this_time,
                 enum "dr_id" *request_dr,
                 enum "dr_id" *file_dr)
/* 根据数据文件的信息更新被请求房间的状态和"dr_id"。传递的参数是指针
   变量 request_room_status 和"dr_id", 以及文件的时间戳和 file_room_status。函
   数根据 file_room_status 的值更新这些值 */
{
    struct timeval current_time;
    /* 用于表明当前时间和核对 pending 的时间是否超过 10 分钟 */
    gettimeofday (&current_time, NULL);
    /* 取得当天的当前时间以得到 pending 状态的时间戳 */
    if (*file_room_status != open)
        /* 如果文件中该时间不可用 (不处于 open 状态), 则用当前的状态更新 re-
           quest。这将告诉用户所选的房间不可用 */
    {
        *request_room_status = *file_room_status;
    } /* 结束 if */
    else if (*file_room_status == open)
        /* 若该房间状态为 open 并且此前 request 中相关信息没有从 open 状态改变,
           则将数组写为 pending, 这表明用户所需求的该时间可用。然后客户有 10
           分钟时间对 request 做出响应或丢弃它。再用持有该时间段的医生名更新文
           件。该名字仅仅允许那个医生访问 10 分钟 */
    {
        *file_room_status = pending;
        *this_time = current_time.tv_sec;
        *file_dr = *request_dr;
    } /* 结束 if */
    return (0);
} /* 结束函数 update_stat */

/***** select_stat *****/

```

```
int select_stat(enum room_status *request_room_status,
               enum room_status *file_room_status,
               enum "dr_id" *request_dr,
               enum "dr_id" *file_dr)
```

/* 如果发送的消息中房间状态为 select，则函数更新这个状态值。传递的参数是指向 room_status 的指针变量，该变量中状态值为 select。函数根据下面的信息更新参数：如果已选择该时间的请求值，则检查数组的状态。如果该 "dr_id" 是该时间的最后一个 ID，则将此时间更新为 busy 状态 */

```
{
    if (*file_dr == *request_dr)
    {
        *file_room_status = busy;
        *request_room_status = confirm;

        | /* 结束 if */
    else
    {
        *request_room_status = busy;

        | /* 结束 else */
    }
    return (0);
    | /* 结束函数 select_stat */
}
```

403

A.8 Common Source Code

A.8.1 Commonh.h

/* commonh.h 文件用于维护一些外科手术调度程序中客户端和服务端公用的头文件和定义。该程序是 John A. Fritz 在 Dr. D. L. Galli 的指导下完成 */

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/time.h>
#define SERVER_HOST "???.??.??"
```

```

    /* 用做医院服务器的主机 IP 地址。注意：必须给它赋值！ */
#define KNOWN_PORT 20000
    /* 与服务器连接的已知端口号 */
enum surgery_type {neuro, cardio, plastic, ortho};
    /* 手术的类型，根据该类型分配相应的房间 */
enum "dr_id" {williams, jones, smith, johnson, master, none};
    /* 允许在医院中进行活动的医生的标志。Master 在服务器端主函数中告诉
    服务器进行关闭操作 */
enum room_status {busy, open, pending, Select, confirm};
    /* 某时间所有可能的状态。时间提交时为 busy；若时间可用则为 open；
    时间已被另一个医生选择时为 pending；Select 用来表示医生根据有效状
    态所选择的时间；confirm 告诉医生医院已经处理并确认了该选择 */
struct initial_connect
    /* 从客户端发出初始请求的结构体 */
{
    enum "dr_id" dr_name;
    /* 发出请求的客户 id */
    int new_port;    /* 从服务器返回到客户端用于进一步通信的端口号。最初发送到
    服务器时它应该初始化为 0 */
};
struct day_request
    /* 在客户和服务器之间传递信息的结构体 */
{
    enum "dr_id" dr_name;
    /* 发出请求的客户 id */
    enum surgery_type surgery_name;
    /* 将安排的手术的名字 */
    int julian_date;
    /* 手术约日期 */
    enum room_status status0800;
    /* 某个特定时间的有效性状态 */
    enum room_status status0900;
    enum room_status status1000;
    enum room_status status1100;
    enum room_status status1200;
    enum room_status status1300;
    enum room_status status1400;
    enum room_status status1500;
    enum room_status status1600;

```

404

405

```

};

int print_day_request (struct day_request request);
    /* 显示 day_request 类型请求信息的函数。传递的参数是结构体类型 day_
       request 的变量。该结构的一些状态将被显示出来 */
int print_status (enum room_status this_room_status);
    /* 显示房间状态的函数。传递的参数是结构体类型 room_status 的变量，
       将显示其中的一些状态 */
int print_dr (enum "dr_id" this_dr);
    /* 显示 dr_name 的函数。传递的参数是枚举类型 "dr_id" 变量 */
int print_type (enum surgery_type this_type);
    /* 显示手术类型的函数。传递的参数是结构体类型 surgery_type 变量 */

```

A.8.2 Commonc.c

/* "commonc.c" 文件包含调度程序中服务器和客户端公用的函数。包括的函数有：

```

print_day_request (), print_status (), print_dr (), print_type () */
#include "commonh.h"
int print_day_request (struct day_request request)
    /* 显示 day_request 类型请求信息的函数。传递的参数是结构体类型 day_
       request 的变量。该结构的一些状态将被显示出来 */
{
    printf("\n\n\n\n Surgeon Name: Dr. ");
    print_dr(request.dr_name);
    printf("Surgery Type: ");
    print_type(request.surgery_name);
    printf("Surgery date: %d \n ", request.julian_date);

    /* 显示房间状态的 shell，将枚举类型 room_status 变量传递到函数 print_sta-
       tus */

    printf("Room Status: \n ");
    printf(" 08:00 - ");
    print_status(request.status0800);
    printf(" 09:00 - ");
    print_status(request.status0900);
    printf(" 10:00 - ");
    print_status(request.status1000);
    printf(" 11:00 - ");
    print_status(request.status1100);
    printf(" 12:00 - ");
    print_status(request.status1200);
    printf(" 13:00 - ");
    print_status(request.status1300);
    printf(" 14:00 - ");
    print_status(request.status1400);
    printf(" 15:00 - ");

```

```

print_status(request.status1500);
printf(" 16:00 - ");
print_status(request.status1600);
return (0);

} /* 结束函数 print_day_request */

int print_status (enum room_status this_room_status)
    /* 显示房间状态的函数。传递的参数是结构体类型 room_status 的变量 */
{
    switch (this_room_status)
        /* 测试 room_status 并显示相关的名字。如果该名字不在列表之中, 这个
           错误并不严重, 程序不会退出。函数将会显示出空白的状态值 */
        {
            case busy: printf("Busy \n");
                        break;
            case open: printf("Open \n");
                        break;
            case pending: printf("Pending \n");
                           break;
            case Select: printf("Select \n");
                           break;
            case confirm: printf("Confirmed \n");
                           break;
        } /* end switch */
    return(0);

} /* 结束函数 print_status */

int print_dr (enum "dr_id" this_dr)
    /* 显示 dr_name 的函数。传递的参数是枚举类型"dr_id"变量 */
{
    switch (this_dr)
        /* 测试 this_dr 并显示相应的名字。如果该名字不在列表之中, 这个错误
           是严重的, 文件需要在退出程序后立即进行更新。显然这需要加强专
           业能力, 在这里为了简化和减小空间而仅仅用了最基本的方法 */
        {
            case smith : printf("Smith \n");
                           break;
            case jones : printf("Jones \n");
                           break;
            case johnson : printf("Johnson \n");
                             break;
            case williams : printf("Williams \n");
                              break;
            case none : printf("None \n");
                          break;
            default: printf("Invalid Dr name \n");
                       exit(1);
        }
}

```

```

    } /* 结束 switch */
    return (0);
} /* 结束函数 print_dr */
int print_type (enum surgery_type this_type)
    /* 显示手术类型的函数。传递的参数是结构体类型 surgery_type 变量 */
{
    switch (this_type)
        /* 测试 surgery_name 并显示相关的名字。如果该名字不在列表之中，这个
           错误是严重的，文件需要在退出程序后立即进行更新 */
        {
            case neuro : printf("Neurological \n");
                          break;
            case cardio : printf("Cardio Vascular \n");
                          break;
            case plastic : printf("Plastic \n ");
                          break;
            case ortho : printf("Orthopedic \n");
                        break;
            default: printf("Invalid Surgery Type \n");
                    exit(1);
        }
    } /* 结束 switch */
    return (0);
} /* 结束函数 print_type */

```

A.9 文件初始化源代码：WRITE_CA.C

/* "write_ca.c" 文件包含初始化外科手术调度程序中服务器所使用的数据文件的函数。程序将建立 room_struct 类型的数组并将信息写回到用户选择的文件中。程序结束后建立相应的文件。该程序是 John A. Ariz 在 Dr. D.L. Galli 的指导下完成 */

```

#include "commonh.h"
    /* 包含所有公共头文件和定义的本地文件 */
#include "hospital.h"
    /* 包含所有公共头文件和定义的本地文件，尤其是为服务器所专用的头
       文件和定义 */

/***** main *****/
void main ()
{
    char user_select [1],
        /* 存放用户的选择 */
        this_file [25];
        /* 存储文件名字符串 */

    int i;          /* 用来遍历数组 */

```

```

struct timeval current_time;
    /* 包含当前时间的结构体 */
FILE *room_file;
struct room_struct init_room [356];
    /* 保存房间有效信息的结构体数组 */
printf ("\n\n\r."Which file would you like to initialize? \n");
do
    /* 循环直到用户要求退出 */
{
    do
        /* 循环直到用户作出有效选择 */
    {
        printf("Please enter the appropriate character: \n ");
        printf("
            A = Neurological Rooms \n
            B = Cardiovascular Rooms \n
            C = Orthopedic Rooms \n
            D = Plastics Rooms \n
            Z = EXIT \n");
        scanf("%s",user_select);

        /* 检索用户输入并将它改成大写形式 */
        user_select[0] = toupper(user_select[0]);
        switch(user_select[0])
            /* 让用户选择要初始化的文件, 并将文件名复制到 this_file。this_file 将用于
            打开文件进行写操作 */
        {
            case 'A': strcpy(this_file,"neuro_rooms.dat");
                break;
            case 'B': strcpy(this_file,"cardio_rooms.dat");
                break;
            case 'C': strcpy(this_file,"ortho_rooms.dat");
                break;
            case 'D': strcpy(this_file,"plastic_rooms.dat");
                break;
            case 'Z': printf("Exiting File Init Program \n");
                break;
            default: printf("This Option is Invalid \n");
                break;
        }
    } /* 结束 switch */
} /* 结束 do-while */
while((user_select[0] != 'A')
    /* 持续循环直到做出选择 */
    &&(user_select [0] != 'B')
    &&(user_select [0] != 'C')
    &&(user_select [0] != 'D')

```



```

    &&(user_select[0] != 'Z'));
if (user_select[0] != 'Z')
    /* 选择必须有效。如果不是'Z'，则打开文件建立数组并进行写操作 */
{
    if ((room_file = fopen (this_file, "w")) ==
        (FILE *) NULL)
    {
        printf(" %s didn't open %d \n",
            this_file, errno);
        exit(1);
    } /* 结束 if */
    for (i=1; 365 >= i; i++)
    {
        init_room[i].julian_date=i;
        /* 填写一年的 julian_date 中要求的手术的日期 (闰年不考虑) */
        init_room[i].status0800=open;
        /* 在某个特定时间的有效性状态最初都初始化为 open，当程序运行时将根据
           当前值进行更新，更新的值还将写回到文件中 */
        init_room[i].status0900=open;
        init_room[i].status1000=open;
        init_room[i].status1100=open;
        init_room[i].status1200=open;
        init_room[i].status1300=open;
        init_room[i].status1400=open;
        init_room[i].status1500=open;
        init_room[i].status1600=open;
        gettimeofday (&current_time, NULL);
        /* 调用函数得到当前时间用来为时间段设置一些值 */
        init_room[i].ptime0800=current_time.tv_sec;
        /* 标志房间被设置成 pending 状态的时间戳。如果时间戳大于 10 分钟，则用在
           那个时间的下一个请求有效消息重新设置该值 */
        init_room[i].ptime0900=current_time.tv_sec;
        init_room[i].ptime1000=current_time.tv_sec;
        init_room[i].ptime1100=current_time.tv_sec;
        init_room[i].ptime1200=current_time.tv_sec;
        init_room[i].ptime1300=current_time.tv_sec;
        init_room[i].ptime1400=current_time.tv_sec;
        init_room[i].ptime1500=current_time.tv_sec;
        init_room[i].ptime1600=current_time.tv_sec;
        init_room[i].dr0800=none;
        /* 特定时间的 dr_id。将所有的值都初始化为 none。当程序运行时用当前的值

```

```
        进行更新并将更新的值写回文件 */
    init_room[i].dr0900 = none ;
    init_room[i].dr1000 = none ;
    init_room[i].dr1100 = none ;
    init_room[i].dr1200 = none ;
    init_room[i].dr1300 = none ;
    init_room[i].dr1400 = none ;
    init_room[i].dr1500 = none ;
    init_room[i].dr1600 = none ;
    fwrite(&init_room[i], sizeof(init_room[i]), 1,
        room_file);
} /* 结束 for */
printf ("\n\n\n %s has been initialized \n", this_file);
printf ("Would you like to initialize another? \n");
fclose (room_file);
} /* 结束 if */
} /* 结束 do-while */
while (user_select [0] != 'Z');
    /* 持续初始化文件直到用户退出 */
} /* 结束 main */
```

缩写词表

AAL (ATM Adaptation Layer)	ATM 适配层
ACID (Atomicity, Consistency, Isolation, Durability (properties of a transaction))	原子性, 一致性, 独立性, 持久性 (事务的特性)
ACL (Access Control Lists)	访问控制表
ACM (Association for Computing Machinery)	计算机协会
ACPI (Advanced Configuration and Power Interface)	高级配置和电源接口
AFS (Andrew File System)	Andrew 文件系统
API (Application Programming Interface)	应用程序设计接口
APM (Advanced Power Management)	高级电源管理
APPN (Advanced Peer-to-Peer Network)	高级对等网络
ARPANET (Advanced Army Research Projects Network)	高级军队研究项目网络
ASCII (American Standard Code of Information Interchange)	美国信息变换标准代码
ATM (Asynchronous Transfer Mode)	异步传输模式
BOA (Basic Object Adapter (CORBA))	基本对象适配器 (CORBA)
BSD (Berkeley Software Distribution)	Berkeley (伯克莱) 软件发布 (版本)
CCITT (Intl. Telegraph & Telephone Consultative Committee (now ITU-T))	国际电报电话协调委员会 (现在是 ITU-T)
CDFS (Compact Disk File System)	密集磁盘文件系统
COM (Component Object Model)	组件对象模型
COOL (Chorus Object-Oriented Layer)	Chorus 面向对象层
CORBA (Common Object Request Broker Architecture)	公共对象请求代理结构
CSMA/CD (Carrier Sensitive Multiple Access with Collision Detection)	载波侦听多路存取/冲突检测
DAP (Directory Access Protocol)	目录访问协议
DARPA (Defense Advanced Research Projects Agency)	国防部高级研究计划局
DCE (Distributed Computing Environment)	分布式计算环境
DCOM (Distributed Component Object Model)	分布式组件对象模型
DNS (Directory Name Service)	目录名服务
DES (Data Encryption Standard)	数据加密标准
DFS (Distributed File System (Windows 2000))	分布式文件系统 (Windows 2000)
DIB (Directory Information Base (X.500))	目录信息库 (X.500)
DISP (Directory Information Shadowing Protocol)	目录信息映射协议
DIT (Directory Information Tree (X.500))	目录信息树 (X.500)
DLL (Dynamic Link Library)	动态链接库

- DNS (Distributed Name System, Domain Name System (or Server)) 分布式名字系统, 域名系统 (or 服务器)
- DoD (Department of Defense) 国防部
- DSM (Distributed Shared Memory) 分布式共享内存
- DSP (Directory System Protocol) 目录系统协议
- DPC (Deferred Procedure Call (Windows 2000)) 延迟过程调用 (Windows 2000)
- DVD (Officially not an acronym, although once stood for Digital Video Disk) 并非正式的简写词, 一般代表数字视频盘
- EBCDIC (Extended Binary-Coded Decimal Interchange Code) 扩充二—十进制交换码
- EDR (External Data Representation) 扩展数据表示
- EFS (Encrypting File System (Windows 2000)) 加密文件系统 (Windows 2000)
- FAT (File Allocation Table) 文件分配表
- FAQ (Frequently Asked Questions) 常见问题
- FDO (Functional Device Objects (Windows 2000)) 功能设备对象 (Windows 2000)
- FDDI (Fiber Distributed Data Interface) 光缆分布式数据接口
- FEK (File Encrypting Key) 文件密钥
- FIFO (First In First Out) 先进先出
- FTP (File Transfer Protocol) 文件传输协议
- GEOS (Geostationary Operational Environmental Satellites) 卫星地面站操作环境
- GSM (Global System for Mobile Communications) 移动通信全球系统
- GUI (Graphical User Interface) 图形用户界面
- HAL (Hardware Abstraction Layer (Windows 2000)) 硬件抽象层 (Windows 2000)
- HTML (Hyper Text Markup Language) 超文本标记语言
- HTTP (Hypertext Transfer Protocol) 超文本传输协议
- ICMP (Internet Control Message Protocol) 网际控制报文协议
- IDL (Interface Definition Language) 接口定义语言
- IETF (Internet Engineering Task Force) 因特网工程任务组
- IGMP (Internet Group Management Protocol) 网际管理协议
- IIS (Internet Information Service) 因特网信息服务
- IP (Internet Protocol) 因特网协议
- IPC (InterProcess Communication) 进程间通信
- IRP (I/O Request Packets) 输入输出请求包
- ISO/OSI (International Standards Organization/Open System Interconnection) 国际标准化组织/开放系统互联
- ITU-T (International Telecommunications Union-Telecommunication (sector)) 国际电信联盟—远程通信部
- IVR (Interactive Voice Response) 交互声音响应
- KDMB (Kerberos Database Management System) Kerberos 数据库管理系统
- LDAP (Lightweight Directory Access Protocol) 轻便目录访问协议

- LPC (Local Procedure Call) 本地过程调用
- MIMD (Multiple Instruction Stream, Multiple Data Stream) 多指令流, 多数据流
- MISD (Multiple Instruction Stream, Single Data Stream) 多指令流, 单数据流
- MMC (Microsoft Management Console) 微软管理控制台
- MMU (Memory Management Unit) 内存管理单元
- NBS (National Bureau of Standards (now NIST)) 国家标准化局 (现是 NIST)
- NCP (Network Control Program) 网络控制程序
- NCSC (National Computer Security Center) 国家计算机安全中心
- NFS (Network File System) 网络文件系统
- NIS (Network Information Services) 网络信息服务
- NIST (National Institute of Standards and Technology) 国家标准化与技术研究所
- NORMA (No Remote Memory Access (multiprocessor)) 非远程内存访问 (多处理器)
- NRU (Not Recently Used) 非最近使用
- NSA (National Security Administration) 国家安全局
- NSF (National Science Foundation (USA)) 国家科学基金会 (USA)
- NSP (Name Server Protocol) 名字服务器协议
- NT (New Technology) 新技术
- NTFS (NT File System) NT 文件系统
- NTP (Network Time Protocol) 网络时间协议
- NUMA (Non-Uniform Memory Access (multiprocessor)) 非一致内存访问 (多处理器)
- OEM (Original Equipment Manufacturers) 原始设备制造厂家
- OMG (Object Management Group) 对象管理组织
- ONC (Open Network Computing (from Sun)) 开放式网络计算 (来自 SUN)
- ORG (Object Request Broker) 对象请求代理
- OSF (Open Software Foundation) 开放软件基金会
- OU (Organizational Units (Windows 2000)) 组织单元 (Windows 2000)
- PAID (Prevent, Avoid, Ignore or Detect (methods for handling deadlocks)) 防止, 避免, 忽略或检测 (处理死锁的方法)
- PDO (Physical Device Objects (Windows 2000)) 物理设备对象 (Windows 2000)
- POA (Portable Object Adapter (CORBA)) 灵活对象适配器 (CORBA)
- PRAM (Pipelined Random Access Memory (a consistency model)) 管道随机存取内存 (一种一致性模型)
- PSTN (Public-Switched Telephone Network) 公共开关电话网络
- RAM (Random Access Memory) 随机存取内存
- RCP (Remote CoPy) 远程复制
- RDN (Relative Distinguished Name (X.500)) 相关识别名 (X.500)
- RFC (Request for Comments) 请求注释
- ROT (Running Object Table (DCOM)) 运行对象表 (DCOM)
- RPC (Remote Procedure Call) 远程过程调用

- RSA (Rivest, Shamir, and Adleman (their public-key encryption algorithm)) Rivest, Shamir 和 Adleman 三种公钥加密算法
- SCSI (Small Computer Systems Interface) 小型计算机系统接口
- SDDL (Security Descriptor Definition Language (Windows 2000)) 安全性描述定义语言 (Windows 2000)
- SDK (Software Developer's Kit (Windows 2000)) 软件开发者工具包 (Windows 2000)
- SIGOPS (Special Interest Group in Operating Systems (ACM)) (ACM) 操作系统专业组
- SIMD (Single Instruction Stream, Multiple Data Stream) 单指令流, 多数据流
- SISD (Single Instruction Stream, Multiple Data Stream) 单指令流, 单数据流
- SMTP (Simple Mail Transfer Protocol) 简单邮件传输协议
- SNMP (Simple Network Management Protocol) 简单网络管理协议
- SNTP (Simple Network Time Protocol) 简单网络时间协议
- SOP (Subject-Oriented Programming) 面向主题程序设计
- SSL (Secure Socket Layer) 安全套接字层
- SSPI (Security Support Provider Interface (Windows 2000)) 安全性支持提供者接口 (Windows 2000)
- TCP (Transmission Control Protocol) 传输控制协议
- TIG (Task Interaction Graph) 任务交互图
- TI-RPC (Transport-Independent Remote Procedure Call) 独立传输远程过程调用
- TNI (Trusted Network Interpretation) 可用网络说明
- UDF (Universal Disk Format) 通用磁盘格式
- UDP (User Datagram Protocol) 用户数据报协议
- UMA (Uniform Memory Access (multiprocessor)) 一致性 (标准) 内存访问 (多处理器)
- USN (Update Sequence Number (Windows 2000)) 更新序列号 (Windows 2000)
- URL (Uniform Resource Locator) 一致的资源定位器
- UTC (Universal Time Coordinator) 调整国际标准时
- VLM (Very Large Memory (64-bit address space)) 超大型内存 (64 位地址空间)
- VLSI (Very Large Scale Integration) 超大规模集成
- VRU (Voice Response Unit) 声音响应单元
- WDM (Win32 Device Manager) Win32 设备管理

术 语 表

- access control (访问控制)** 防止对计算机资源与文件进行未授权访问与损害。
- access list (访问控制表)** 这是一种访问控制的方式, 每个文件或资源都维护一个列表, 其中记录了允许访问此资源的对象以及允许访问的种类。
- access matrix (访问矩阵)** 用来保存系统访问控制表信息的二维数组。
- ACID Property (ACID 特性)** 事务的特性。从观察的角度看, 保证了事务的正确处理。ACID 特性包括原子性、一致性、独立性和持久性, 最早在 [HaRe83] 中提到。
- acquire access (获得访问权)** 释放一致性中所使用的同步原语, 执行读操作时强制本地系统更新共享数据的副本。
- acyclic directory structure (非循环目录结构)** 一种目录结构, 其中一个文件或目录不可以有多个父文件或目录, 这样在结构中就不可能存在环路。
- adaptive scheduler (自适应调度)** 进程在一个特定的位置开始执行后, 因为要适应系统的一些要求, 允许对它进行重新调度的调度规则。所以, 这样的调度也允许进程在执行中的任何时刻进行迁移。
- address space (地址空间)** 整个系统内存的一部分, 用来分配给特别的进程。
- application layer (应用层)** ISO/OSI 参考模型中的第七层。这一层不仅关注用户接口, 并且关注建立基于网络应用的权限与可用性。
- assignment graph (分配图)** 用来表示分布式调度中可能进程分配情况的图。
- asymmetric encryption (非对称加密)** 包含两个密钥的加密。一个密钥对明文进行加密, 另一个密钥用来对密文进行解密。
- asynchronous (异步)** 不在定时间隔发生的事件, 通常由外部来源产生。
- asynchronous primitive (异步原语)** 参见非同步原语。
- atomic action (原子操作)** 一系列不可见的操作, 看上去好像是单个操作。
- atomicity property (原子属性)** 事务的特性, 要求事务中所有的操作在分布式系统中都表现为单个操作。
- authentication (认证)** 检验身份的动作。
- automaton vector (自动机向量)** 表示分布式系统的多种综合负载状态的向量。这个向量用做分布式调度的随机学习。
- backbone (主干网)** 将多个网络连接在一起的网络。一般来说, 在主干网上的“结点”仅为其他的网络。
- barrier (栅栏)** 用做同步以提供释放一致性。这用以区别执行的不同阶段。
- bastion host (设防主机)** 向 Internet 开放的暴露网关机器。它是外界接触的主要结点, 也是系统最容易遭受攻击的位置。它是由加强装备的中世纪城堡外墙命名的。
- block caching (块高速缓存)** 以块为单位远程复制文件部分。
- blocking primitive (阻塞原语)** 一种消息传递原语, 在原语结束操作之前, 内核不会将控

- 制权返回给进程。这也称为同步原语，因为它导致通信的进程相互同步。
- bridge (网桥)** 连接多个实现相同网络协议的网络的设备。只有需要通过特定网络的数据包才会转发到这个网络上。
- bus-contention (总线争用)** 因为多个处理器使用同一总线访问共享内存中的信息而导致总线的过度使用，从而引起系统的效率和效果降低。
- busy wait (忙等待)** 由进程（自旋）执行的连续循环，等待变量值（自旋锁）的改变。
- capability (权能)** 资源的合法用户对资源的一种表示，代表了上述使用权的权限。
- casual consistency model (偶然一致性模型)** 一致性模型之一，要求所有的偶然相关事件对分布式操作系统中所有成员看来都是相同顺序的。
- casual related event (偶然相关事件)** 修改同一集合共享数据的事件。
- certificate authority (认证部门)** 负责发放和检验数字证书的部门，相当于电子化的管理护照的美国护照处或者管理驾照的州汽车局。
- certificate list (认证证书列表)** 提供认证的一种方法。列表包含了不同用户的证书。服务必须核对此列表以检验公钥。
- child process allowance (子进程执行准许)** 处理孤儿进程的一种方法，其中子进程必须不断请求继续准许执行。如果父进程死亡了，它就不会再分发时间允许，这样子进程就会终止执行。
- ciphertext (密文)** 加密或掩饰后的明文。
- change journal (修改日志)** 记录并跟踪对文件和目录内容修改的方法。
- class (类)** 对象的抽象定义。
- clock skew (时钟偏移)** 两个时钟之间因为偏移而产生的差异。
- cluster (集群)** 可度量服务的一种抽象，允许有关的位置相互独立，而且本身可以通过在同等或多个物理服务器上执行所需要的服务来提供冗余和容错。
- communication deadlock (通信死锁)** 包含通信缓存的死锁。
- compound transaction (复合事务)** 包含一个以上子事务的事务。子事务必须组织为层次结构。复合事务和所有嵌套的子事务必须各自及共同支持 ACID 特性。
- computationally secure (计算安全)** 使用利用不可用资源进行的系统分析进行破解的算法对消息进行加密。
- concurrent-copy (并发复制)** 内存迁移的一种方法，复制整个虚拟内存以及其它有关迁移进程的信息，同时在原来的位置并发执行这个进程。
- condition variable (条件变量)** 监视器中的全局变量，用来控制管程中要激活的进程，并确保只有这个进程才是活动的。
- consistency (一致性)** 试图维护数据多个副本的进程，这些数据在多个位置被操作，每个位置都有可接受的数据副本。
- consistency model (一致性模型)** 分布式系统中所有的成员，包括用户应用程序都要遵循的一组定义好的一致性规则。
- consistency property (一致性属性)** 事务的特性，要求根据系统定义的一致性模型，所有事务都要在一致的状态下离开系统。
- constructor (构造器)** 在基于对象的开发中，如果方法有多个实例，那么其中一个就称为

构造器。实例不必为独一无二的。即允许所谓的方法过载。

copy-on-reference (引用时复制) 内存迁移的一种方法，仅在迁移目的地引用到迁移进程虚拟内存的页面时复制这些页面的副本。

copy set (复制集) 用在分布式共享数据内存管理的方法中，维护一个系统中所有参与人员的列表，其中记录了谁持有数据的特定块的副本。

critical region (临界区) 访问共享资源的程序部分。

cryptography (密码学) 产生安全密文的学问。

cutset (割集) 边的集合，如果去掉了这些边，连通图将成为不连通图，即有两个或更多不能通过边连接的组件。

cyclic directory structure (循环目录结构) 一种目录结构，其中一个文件或目录可以有多个父文件或父目录，这样在结构中就可能存在环路。而且，子文件或目录可以成为其父文件或目录之一的父文件或目录，这样就产生了循环结构。

data link layer (数据链路层) ISO/OSI 参考模型中的第二层。这一层可以分为两个子层：媒体访问控制和逻辑链路控制。

deadlock (死锁) 资源分配的问题，导致分配方案禁止两个或更多的进程取得进展。

decryption (解密) 将合适的算法和合适的密钥作用在信息上，使其可读。

digest function (摘要函数) 函数的输入是文档，输出是一个惟一值，用做数字签名。这些函数也被称为散列函数。

digital certificate (数字证书) 一种电子的、可验证的标识项，相当于电子的护照或者驾照。

digital signature (数字签名) 在分布式环境中确保初始标识或者内容完整性的技术。

directory service (目录服务) 部分分布式文件系统提供文件系统的用户界面，并对系统的总体结构负责。

direct store-and-forward deadlock (直接存储转发死锁) 正好涉及两个位置的通信死锁。

dispersion (散布值) 指本地时钟相对于 NTP 一级参考时钟允许的最大错误值。散布值越高，表示时钟信息越不可靠。

distributed system (分布式系统) 通过网络连接的多个异质系统统一表现为一个系统及其资源。

divisible process (可分割进程) 可以根据进程特性分割为更小子进程或者任务的进程。这些任务可以独立调度。

domino effect (多米诺效应) 事务中止时如果有其他事务错误地依赖于中止事务而产生的结果。这种情况下，所有其他事务以及依赖于这些事务的事务都要中止。

drifting (偏移) 一度同步的时钟发生的逐渐偏差。

dual-homed host (双重位置主机) 连接到至少两个网络上的计算机系统。

durability property (持久性属性) 事务的特性，要求已经提交的服务器即使发生了系统错误也要完成事务。事务的所有结果都必须是永久的。

dynamic binding (动态绑定) 当进程的可执行影像被创建时并不绑定在特定位置上执行。

economic usage point (经济使用点数) 分布式调度中使用点数方法的一个变种，其中点数表示基于系统负载和其他特性的远程执行价值。

- embedded real-time system (嵌入式实时系统)** 安装在更大系统中的实时系统, 用于控制特别的硬件。
- encryption (加密)** 将合适的算法和合适的密钥作用在信息上, 使其不可读。
- entry release consistency (入口释放一致性)** 基于释放一致性的一致性模型, 同样要求所有的变量均为同步变量, 或者与同步机制结合, 如锁或者栅栏。
- executive (执行程序)** 在 Windows NT 内核中实现系统服务。
- external fragmentation (外部碎片)** 物理内存中在多个内存单位 (如段) 之间所浪费的内存空间。
- explicit addressing (显式寻址)** 消息传递的寻址机制, 其中调用进程指定要通信的确切进程。
- false sharing (错误共享)** 在分布式系统中的多个位置, 当一个页面发生颠簸, 但是没有共享这个页面上的任何数据结构时这种情况就会发生。
- file access semantic (文件访问语义)** 当共享文件发生修改时, 对分布式系统中成员进行通知的一种方法。
- file handle (文件句柄)** 给文件用户的信息, 有文件的位置, 还可能还包括访问上述文件的权限。
- file service (文件服务)** 分布式文件系统的一部分, 负责操作特定的文件。
- file system (文件系统)** 操作系统的一个重要的子系统, 负责维护目录、文件以及管理命名空间。
- firewall (防火墙)** 被设计以防内部计算机资源受到外部的威胁, 提供访问控制。
- firm-real-time system (稳固实时系统)** 能容忍小概率的系统故障的实时系统。
- flow control (流控制)** 这个术语用来形容网络的功能, 它规定了所传输数据的数量和传输的频率, 以防止“压倒”网络资源。
- freeze time (冻结时间)** 在进程迁移过程中, 进程停止和重新启动之间所浪费的执行时间。
- function overloading (功能超载)** 如果允许对一个功能进行多重定义, 所有的这些功能使用完全相同的名称, 但是它们的参数列表和行为不同。
- gang scheduling (集体调度)** 调度的策略, 允许所有相关的进程/线程同时调度。
- global naming (全局命名)** 在分布式系统中, 每个文件或者对象都有独一无二的名称的系统。全局命名不提供命名透明。
- global scheduler (全局调度)** 调度负责在分布式系统中为处理器分配进程。
- global order (全局顺序)** 可以在分布式系统中惟一地标识所有事件顺序的概念。
- global time (全局时间)** 让分布式系统中所有时钟都同步的概念。
- happen-before relationship (超前关系)** 分布式环境中用来获得相对偏序次序的关系。它用来标识偶然关系, 首先由 Lamport [Lam78] 提出。
- hard real-time system (硬实时系统)** 必须满足限定的边界响应时间约束或者能够经受严重的, 可能是有生命威胁后果的系统。
- hash function (散列函数)** 参见摘要函数。
- heuristic (启发规则)** 一般能够达到所需结果的规则。这些规则无需提供的或者可以直接计量的, 但是总能够以有效的方式达到满意的结果。

hierarchical directory structure (层次目录结构) 一种类似树的目录结构，其中给定的文件或者目录只能有一个父级。

host (主机) 连接到网络上的任何计算机系统。

immediate notification (立即通知) 修改通知的一种方法，其中每次对文件的修改都要立即通知分布式系统的参与人员。

immutable file (不可更改文件) 内容不能修改的文件，即不允许修改。

implicit addressing (隐式寻址) 消息传递的寻址机制，其中调用进程指定要通信的服务，代替显式指定进程的名称。

incidental ordering (随机排序) 多对多消息排序的一种形式，仅要求随机相关的消息以正确的顺序接收。

indirect store-and-forward deadlock (非直接存储转发死锁) 包含两个以上位置的通信死锁。

indivisible process (不可分进程) 不能再进一步分解成子进程或者任务的进程，必须在单个处理器上运行。

inheritance (继承) 允许一个对象按照其他对象的形式定义自身。

internal fragmentation (内部碎片) 内存中的块所浪费的内存。

interprocess communication (进程间通信) 两个或更多的进程为了达到共同的目标进行通信的一种方法。

interrupt (中断) 进程向操作系统发出信号，要求某种服务的一种方法。

isochronous (等时) 异步事件的一个子类，事件可以不在精确的时间间隔处发生，这样的话它们就应归为同步事件了，但是发生在一个给定的时间范围内，可以因为网络延迟等原因而稍有偏差。

isolation property (独立特性) 事务的特性，要求事务以“所有或无”的方式来运行，其他的成员不能访问中间结果。

ISO/OSI reference model (ISO/OSI 参考模型) 比较了不同的分布式/网络环境，作为基本系统的标准而建立的七层参考模型。

job (作业) 执行中的程序。这个术语通常指进程的系统视角。

kernel (内核) 操作系统中的特权部分，对于所有的资源都有完全的访问权限。

lazy release consistency (懒释放一致性) 仅在需要时才执行写或更新操作的释放一致性。

leap second (跳秒) 为了确保 UTC 时间，一种基于原子的时间，与天文时间保持一致，有时需要在某一分钟内增加或者减少一秒。

load balancing (负载平衡) 一种负载分配策略，尽量通过迁移保持系统内公平的负载。

load sharing (负载共享) 一种负载分配策略，通过迁移协助工作过度的资源。

local scheduler (本地调度) 负责调度进程到特定位置的实际 CPU 的调度程序。本地调度进程负责在一个地方进行调度，而不涉及在远程位置的调度中。

location resolution (位置解析) 从全局文件名称到文件位置的映射。

logical clock (逻辑时钟) 用来决定事件的相对次序的时钟。逻辑时钟不关心“人类”的时间。

logical link control (逻辑链路控制) ISO/OSI 参考模型中数据链路层的子层。主要的功能包

括错误控制和流控制。

lost update (遗失更新) 如果没有自动地处理事务, 可能在共享数据中发生的潜在问题。它包括多个事务全都复制同一个共享数据来改变它的值。当结果返回时, 每个返回值都覆盖了它的前一个事务的结果。这样, 前一事务的结果就完全丢失了。

mandatory scheduler (强制性调度) 要求分布式系统中的每个位置都参与分布式调度的分布式调度程序, 允许远程进程在本地执行。

marshalling (整理) 在远程过程调用中所执行的函数名称, 把数据整理为在网络上快速通信的格式。

maximum flow of a graph (最大流图) 图包含最大权值的边的子图。

medium access control (媒体访问控制) ISO/OSI 参考模型中数据链路层的子层。这个以局域网为中心的子层主要关心共享广播网络的合理使用。

memory management unit (内存管理单元) 虚拟内存管理器, 将虚拟内存映射为物理内存和磁盘。

message digest (消息摘要) 对文档使用摘要函数的结果。这个摘要可以用来验证文档的内容是否被修改。

message passing (消息传递) 进程间通信的一种方式, 允许两个可能位于不同的机器上进程通过传递拥有共享数据的消息进行通信。

middleware (中间件) 运行在本地或者基本中心操作系统之上的分布式操作系统。

migration (迁移) 在另一个可能位于远程位置处理器上重新载入进程以及所有有关的信息的动作。

min-cut (最小割集) 图的最小割集, 即带有最小代价的割集。割集的代价是由割边权值的总和决定的。

modification notification (修改通知) 用来通知分布式系统的成员分布式系统中的文件已经被修改。

moniker (标记) DCOM 中的对象实例, 标记的接口是 Imoniker。

monitor (管程) 使编译器支持互斥的语言构造。管程是一种抽象的数据类型, 包括变量、过程和数据结构。

multicast (组播) 向所有连接到网络的用户中的一个子集发送信息。

multicomputer (多计算机) 每个处理器都有自己的本地内存的并行计算机。

multiple inheritance (多重继承) 允许一个对象按照其他多个已存在对象的形式定义自身。

multi-threaded (多线程) 允许进程并发执行多个计算线程的环境。

multi-processor (多处理器机) 所有处理器共享同一内存的并行计算机。

mutual exclusion (互斥) 保证多个进程不能同时使用共享资源。

named pipes (命名管道) 使用相关进程的进程间通信形式。

name resolution (名称解析) 从人类(基于字符的)文件名称到全局(二进制)名称的映射。

name service (名字服务) 文件系统的一部分, 负责将文件名映射到文件位置。在分布式环境中, 它可能还要负责将本地名字映射为全局名字, 以及将计算机生成的名字映射为对人类友好的名字。

Name space (名字空间) 系统中文件、目录和组件可使用的名称的完整集合。因为给定系统的名字服务实现的名字规则的缘故，这个集合经常是有限的。

nested transaction (嵌套事务) 参见子事务。

network layer (网络层) ISO/OSI 参考模型中的第三层，它负责路由。对局域网来说，这层是空的。

network system (网络系统) 连接到网络上的异质计算机的集合。系统之间的异质是公开的，对于参与人员的计算机有很少的限制。

non-blocking primitive (非阻塞原语) 一种消息传递原语，其中内核马上将控制权返回给进程，而并不等待到原语结束。它也被称为异步原语。

nonce (临时标签) 用在认证问题中的随机整数值。如果接收到，则返回申请人的消息中必须包含正确的临时标签值。

non-repudiation (非否认) 保证成员不能否认其签名以及签名后的数字信息。

NORMA multiprocessor (NORMA 多处理器) 多处理器，其中每个处理器都有各自的本地内存，处理器之间不能共享内存。

notification on close (关闭时通知) 修改通知的一个类型，当分布式系统的参与人员关闭了它们修改的文件时，其他的成员才会被通知。

notification on transaction completion (事务完成时通知) 修改通知的一个类型，系统使用了事务，并且分布式系统的成员只有在成员完成了事务以后才会被通知。

NP-Hard (NP-困难) 非常难以计算的问题，还没有已知存在的多项式时间算法的解决方案。

NUMA multiprocessor (NUMA 多处理器) 带有共享内存的多处理器，访问共享内存的时间因为它与处理器子群的关系而有所不同。

object (对象) 封装了所有与服务与数据相关的开发抽象。

object-based system (基于对象的系统) 使用对象设计与定义的系统。

object behavior (对象行为) 由特定对象所能够执行的函数的完整集合定义。

object-oriented system (面向对象系统) 使用对象设计与定义，并允许对象继承的系统。

one-time assignment scheduler (一次分配调度) 在进程已经被设计到某特定位置执行后，不允许涉及或迁移进程的调度程序。

optimal scheduling algorithm (最优调度算法) 按照某种标准，有可能确定最好可能的解决方法调度算法。

orphan clean up (孤儿清除) 处理孤儿进程的一种方法，当系统重新启动时，死去父进程的系统终止孤儿进程。

orphan process (孤儿进程) 一个进程在分布式系统中远程执行，而它的父进程已经过早死亡了，可能是因为系统的崩溃。

packet (包) 因特网上通信的主要单位。消息要分解成为包。

packet-filtering gateway (包过滤网关) 有选择地控制或者“过滤”进入或离开内部网络的数据的防火墙。它试图保护网络免遭攻击。

page (页面) 将内存块分割成大小相同的块的概念。分页系统可能会有内部碎片，但不会有外部碎片。

page fault (页面错误) 当一个进程要求一个不在物理内存中的内存的页面时就会发生页面错误。页面错误将导致系统从磁盘或者另一台通过网络连接到分布式系统中的主机上取回页面。

page scanner (页面扫描器) 系统的端口监控程序,用在共享内存模型中监视内存中的页面被非本地访问的频率。

parallel system (并行系统) 异质的处理器和内存模块存在于单独的计算机之内。

peer-to-peer layer (对等层) 这些 ISO/OSI 参考模型层次的功能只在源位置和目标位置执行。

pipe (管道) 当进程至少共享同一个文件系统时,使用进程间通信的一种形式。

physical clock (物理时钟) 用来报告并维护“人类”时间的时钟,与物理时钟相对应。

physical layer (物理层) ISO/OSI 参考七层模型的最低层。这一层涉及诸如如何将位放到网络上以及决定连接物理接口的问题。

plaintext (明文) 想要隐瞒的信息。

plug and play (即插即用) 要求硬件和软件支持并允许连接到系统的计算机设备不需要用户的干涉和配置就能改变并认出。

point-to-point layer (点对点层) ISO/OSI 参考模型中这些层次的功能不仅在源位置和目标位置执行,而且在源位置和目标位置之间上千个可能的网络位置上执行。

polled-loop scheduler (轮询调度) 要求调度程序先询问进程是否需要系统服务的调度程序。这种请求以循环方式操作。

PRAM consistency model (PRAM 一致性模型) 管道随机访问内存一致性模型,要求一个进程所执行的所有写操作要以它们所操作的顺序在其他的进程前出现。

presentation layer (表示层) ISO/OSI 参考模型的第六层。这一层处理信息的语法,包括 ASCII 到 EBCDIC 的转换、压缩、加密等。

private key cryptography (私钥加密) 加密和解密使用一个密钥,它必须被秘密保存。

process (进程) 执行中程序的抽象。

process migration (进程迁移) 将一个进程以及所有相关的状态信息重新装入到另一个处理器上,可能位于远程位置上。

process version number (进程版本号) 处理孤儿进程的一种方法,子进程总是伴随着一个版本号。当父系统重新启动时,它使用一个新的版本号,并向分布式系统的成员通知这个新的版本号。任何执行过期的版本号的进程将会终止。

processor consistency mode (处理器一致性模型) 要求 PRAM 一致以及内存一致的一致性模型。

protected variable (保护变量) 按照释放一致性操作的变量。

proxy-service firewall (代理服务防火墙) 对于使用外部网络的服务,用更加安全的版本来代替原有的服务;特别是它对于外部的服务器来讲是代表内部客户出现的,而不是让客户直接与服务器接触。有助于防止危害网关的后果。

public key cryptography (公钥加密) 存在两个密钥,一个公钥一个私钥。如果一个密钥对消息进行加密,那另一个密钥就可以将其解密。私钥必须秘密保存。

race condition (竞争条件) 输出结果依赖于输入值到达的确切时间。

reactive real-time system (响应实时系统) 对环境不断地交互作出反应的实时系统。

real-time system (实时系统) 必须满足边界响应时间限制的系统, 否则将遭受严重的后果 (参见软实时与硬实时)。

release access (释放访问权) 释放一致性中完成最后一个写操作后所使用的同步原语。它强制向整个系统发布共享数据的改动。

release consistency model (释放一致性模型) 使用获得访问权和释放访问权同步原语的一致性模型。

remote access (远程访问) 一种分布式文件服务, 客户并不在本地传输文件, 而是通过实际的文件服务器进行所有的文件访问。

remote copy (远程复制) 一种分布式文件服务, 客户向他们的位置传输文件的副本。

remote procedure calls (RPC) (远程过程调用) 进程间通信的一种方式, 使用相似过程的格式。这种方法也允许被调用的过程返回一个值, 而不仅仅是一个确认。

reparse tag (再解析标记) 指示已经到达了再解析点, 并且适用于给定的文件。

reparse point (再解析点) 通过使用文件名和再解析标记中所指定的过滤器, 让文件系统过滤器改变文件处理方式的一种办法。

retrieval disparity (检索不一致) 如果没有自动地处理事务并保持独立性, 可能在共享数据中发生的潜在问题。其中, 取回的数据值不能反映最近事务的结果。

repeater (中继器) 连接两个实现相同协议网络的设备。一个网络上的每个数据包都直接转发到另一个网络。

request (请求) 在基于对象的开发中, 一个对象调用另一个对象服务的方法。

router (路由器) 连接两个实现了不同协议的设备。

session layer (会话层) ISO/OSI 参考模型的五层, 负责源位置和目标位置之间通信的同步。

segment (段) 一种内存块, 大小可变动。段会有外部碎片, 同时也会有少量的内部碎片。

sequential consistency model (顺序一致性模型) 要求所有参与人员共享有关事务顺序相同的全局视图的一致性模型。

shared memory model (共享内存模型) 分布式计算环境中内存管理的模型, 进程通过通用数据结构进行通信。

simple memory model (简单内存模型) 并行 UMA 架构所使用的内存管理模型, 直接基于集中式系统的内存管理。

snoopy cache (窃听高速缓存) 一种高速缓存一致性模型, 其中处理器监视共享总线的有关修改当前其本地高速缓存副本的数据的信息。

stateful file service (有状态文件服务) 一种分布式文件服务类型, 其中文件服务器维护所有客户和他们正在访问的文件列表的状态信息, 还有他们在给定文件中的位置。

stateless file service (元状态文件服务) 一种分布式文件服务类型, 其中文件服务器并不维护任何客户的状态信息。所有的客户请求都必须带有有关它们特别请求的完整信息。

static binding (静态绑定) 创建进程的可执行影像时, 进程马上与执行的具体位置进行绑定。

status state (状态情况) 分布式系统中表示一个特定系统负载的值。有些实现方式使用了两个状态: 欠载和过载, 而其他方式则还使用了合理负载作为第三种状态。

stratum (层次) NTP 中所使用的术语, 指的是客户或者服务器在树状体系结构中所处的等级。

stochastic learning (随机学习) 分布式调度的一种方法, 其中算法从先前的选择中得到反馈, 然后将这个信息用来促进以后的选择。

strict consistency model (严格一致性模型) 要求所有读操作从最近的写操作处返回值的一致性模型。

structured file (结构化文件) 文件存储的一种方法, 每个文件都分解成多个记录。文件系统通过外部索引支持这些记录。

socket (套接字) 进程间通信的一种方法, 要求通信进程绑定在公用的端点上。

soft real-time system (软实时系统) 必须满足边界响应时间约束的系统, 否则将遭受如性能下降的不良后果。

spinlock (自旋锁) 由循环或自旋不断检查的变量。自旋锁常用于忙等待。

spinning (自旋) 忙等待中不断检查变量的值的循环。

stop-and-copy (停止和复制) 停止执行的内存迁移的一种方法, 将与迁移进程相关的实体虚拟内存复制, 然后在迁移目的地重新开始执行。

sub-optimal approximate scheduling algorithm (次优近似调度算法) 使用最优方法的“捷径”来快速获得次优方法的调度算法。

sub-optimal heuristic scheduling algorithm (次优启发式调度算法) 使用基本直觉考虑规则的调度算法往往会得到更好的整体系统性能。这些规则不是必须提供的或者可以直接衡量对系统性能的作用的。

sub-optimal scheduling algorithm (次优调度算法) 试图找出按照某种标准来衡量的非常好的解决方案的调度算法, 但不必是最优解。次优算法比最优算法要快得多。

sub-transaction (子事务) 庞大、复合事务的一部分事务, 也被称为嵌套事务。

symmetric encryption (对称加密) 涉及单个密钥的加密。使用同一个密钥对明文进行加密, 对密文进行解密。

synchronization variable (同步变量) 用来同步内存与维护弱一致性。

synchronous (同步) 发生于精确、规则、可预计时间间隔的事件, 一般从实时系统的附加组件处产生。

synchronous primitive (同步原语) 参见阻塞原语。

Task Interaction Graph (任务交互图) 图的节点表示相关的进程, 而边表示进程之间的交互。这种图用来表示可以模块化划分的负载。

thrashing (颠簸) 内存管理中的术语, 形容系统长期处于取出并移走内存页面的状态, 导致了性能的严重下降。

thread (线程) 执行线程的简称, 表示贯穿程序的一条路径。

time provider (时间提供者) 能够直接从 UTC 服务器上接受信息, 并作出适当调整以弥补通信延迟的商业设备。

transaction (事务) 共享数据上的归为一组的一组操作, 作为单一动作看待。

transaction management (事务管理) 为了确保事务的正确操作而使用的服务。

transport layer (传输层) ISO/OSI 参考模型的第四层。这一层建立了通信服务种类。服务

的种类可以是面向连接的或者无连接的。

two-phase commit protocol (两阶段提交协议) 实现事务的一个协议。这个协议分成准备提交阶段和提交阶段，遵循事务的 ACID 属性。

unnamed pipe (未命名管道) 进程间通信的一种方式，不相关的进程共享一个公用文件系统时可以使用。

UMA multiprocessor (UMA 多处理器) 一种多处理器，其中所有的处理器共享内存，而访问任何内存模块的访问时间是均匀的（相等的）。

uniform ordering (均匀排序) 一种多对多消息排序的形式，要求所有的消息均匀地按照所有接收进程进行接收，参与人员接收的顺序可以与发送的顺序不同。

universal number (统一号) 不依赖于地区码的电话号码，表现出位置的独立性。

universal ordering (通用排序) 一种多对多消息排序的形式，要求并且保证所有的消息严格按照发送的顺序被接收。

update sequence number (更新序列号) 修改日志中表示对文件或目录进行特别修改的数字。

usage point (使用点数) 分布式调度的一个方法，为使用外部资源的计算机支取点数，而对允许远程位置使用其本地资源的计算机增加信用。

usage table (使用表) 集中式服务器上的一个位置，维护分布式系统中所有成员运行时使用的点数总和。

very large memory (超大内存) 地址空间能够支持 64 位的地址。

virtual memory (虚拟内存) 一个内存管理的概念，允许进程相信其拥有自己所需要的所有内存，即使这部分内存的数量大于实际的物理内存。

voluntary scheduler (自愿调度) 不强制分布式系统中的每个位置都要允许远程进程在本地执行的调度算法。分布式调度中的参与人员是完全自愿的。

weak consistency model (弱一致性模型) 使用同步变量的一致性模型。当使用了同步变量之后，所有的本地写操作都要传播到整个系统，而每个位置上的系统都要更新自己所持有副本的数据。

whole file caching (全文件高速缓存) 一种分布式文件服务，其中客户向他们的本地位置传输文件的副本，也被称为远程复制。

参考文献目录

- [AAO92] Abrossimov, A., F. Armand, and M. Ortega. "A Distributed Consistency Server for the CHORUS System." *Proceedings of SEDMS III, Symposium on Experience with Distributed and Multiprocessor Systems*. USENIX, pp. 129-148: 1992.
- [ABCLMN98] Anderson, R., F. Bergadano, B. Crispo, J. Lee, C. Manifavas, and R. Needham. "A New Family of Authentication Protocols." *Operating Systems Review*. ACM SIGOP Press. Vol 32, Num 4, pp. 9-20: 1998.
- [ABLL91] Anderson, T., B. Bershad, E. Lazowska, and H. Levy. "Scheduler Activations: Effective Kernel Support for the User Level Management of Parallelism." *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 53-79: 1991.
- [AbPo86] Abrams, M. and H. Podell. *Tutorial: Computer and Network Security*. IEEE Computer Society Press., Washington D.C., 1986.
- [ADFS98] Arbaugh, W., J. Davin, D. Farber, and J. Smith. "Security for Virtual Private Intranets." *IEEE Computer*. Vol. 31, No. 9, pp. 48-55: 1998.
- [AdHi90] Adve, S. and M. Hill. "Weak Ordering: A New Definition." *Proceedings of the 17th IEEE Annual International Symposium on Computer Architecture*. pp. 2-14: 1990.
- [AgAb91] Agarwal, D. and A. El Abbadi. "An Efficient and Fault-Tolerant Solution of Distributed Mutual Exclusion." *ACM Transactions on Computer Systems*. Vol. 9, pp. 1-20: 1991.
- [AGGL86] Armand, F., M. Gien, M. Guillemont, and P. Leonard. "Towards' a Distributed UNIX System - The CHORUS Approach." *Proceedings of the EUUG Autumn '86 Conference*. Manchester, England, Autumn 1986.
- [AGS83] Ames, S., M. Gasser, and R. Schell. "Security Kernel Design and Implementation: An Introduction", *IEEE Computer*. Vol. 16, No. 7, pp. 14-22: 1983.
- [AJP95] Abrams, M., Jajodia, and H. Podell (Eds). *Information Security: An Integrated Collection of Essays*. IEEE Computer Society Press., Washington D.C., 1995.
- [Akl97] Akl, S. *Parallel Computation: Models and Methods*. Prentice Hall, Englewood Cliffs, New Jersey, 1997.
- [AKZ96] Awad, M., J. Kuusela, and J. Ziegler. *Object-Oriented Technology for Real-Time Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1996.
- [And94] Anderson, R. "Why Cryptosystems Fail." *Communications of the ACM*. Vol. 37, No. 11, pp. 32-40: 1994.
- [ArFi89] Artsy, Y., and R. Finkel. "Designing a Process Migration Facility." *IEEE Computer*. Vol. 22, No. 9, pp. 47-56: 1989.
- [ARJ97] Anderson, J., S. Ramamurthy, and K. Jeffay. "Real-Time Computing with Lock-Free Shared Objects." *ACM Transactions on Computer Systems*. Vol. 15, No. 2, pp. 134-165: 1977.
- [ARS89] Abrossimov, V., M. Rozier, and M. Shapiro. "Generic Virtual Memory Management for Operating System Kernels." *Proceedings of the 12th ACM Symposium on Operating Systems Principles*. Pp. 123-136: 1989.

- [BaHa87] Bacon, J. and K. Hamilton. *Distributed Computing with RPC: The Cambridge Approach*. The University of Cambridge Computer Laboratory Technical Report No. 117, England: 1987.
- [BAN90] Burrows, M., M. Abadi, and R. Needham. "A Logic of Authentication." *ACM Transactions on Computer Systems*. Vol. 8, No. 1, pp. 18-36: 1990.
- [BALL90] Bershad, B., T. Anderson, E. Lazowska, and H. Levy. "Lightweight Remote Procedure Call." *ACM Transactions on Computer Systems*. Vol. 8, No. 1, pp. 37-55: 1990.
- [BBF83] Bauer, R., T. Berson, and R. Feirtag. "A Key Distribution Protocol Using Event Markers." *ACM Transactions on Computer Systems*. Vol. 1, No. 3, pp. 249-255: 1983.
- [BCEF94] Bauer, M., N. Coburn, D. Erickson (Galli), P. Finnigan, J. Hong, P. Larson, J. Pahl, J. Slonim, D. Taylor, and T. Teorey. "A Distributed System Architecture for a Distributed Application Environment." *IBM Systems Journal*. Vol. 33, No. 3, pp. 399-425: 1994.
- [BeFe91] Berthome, P. and A. Ferreira. "On Broadcasting Schemes in Restricted Optical Passive Star Systems." *Interconnection Networks and Mapping and Scheduling Parallel Computations*. D. Hsu (ed.) DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 21, pp. 19- 30: 1991.
- [Ber92] Berson, A. *Client/Server Architecture*. McGraw Hill, New York, New York, 1992.
- [BeZe91] Bershad, B. and M. Zekauskas. *Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors*. Carnegie Mellon University Technical Report: CMU-CS-91: 1991.
- [BFMV84] Blake, I., R. Fuji-Hara, R. Mullin, and S. Vanstone. Computing Logarithms in Finite Fields of Characteristic Two. *Siam Journal of Algorithms for Discrete Mathematics*, pp. 276-285, 1984.
- [BGGLO94] Bricker, A., M. Gien, M. Guillemont, J. Lipkis, D. Orr, and M. Rozier. "A New Look at Micro-kernel Based UNIX Operating Systems: Lessons in Performance and Compatibility." *Distributed Open Systems*, IEEE Computer Society Press, Los Alamitos, California, pp. 31-48: 1994.
- [BGMR96] Bharadwaj, V., D. Ghose, V. Mani, and T. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, California, 1996.
- [BiJo87] Birman, K. and T. Joseph. "Reliable Communication in the Presence of Failures." *ACM Transactions on Computer Systems*. Vol. 5, No. 1, pp. 47-76: 1987.
- [Bir85] Birrell, A. "Secure Communication Using Remote Procedure Calls." *ACM Transactions on Computer Systems*. Vol. 3, No. 1, pp. 1-14: 1985.
- [BKT85] Brown, M., K. Kolling, and E. Taft. "The Alpine File System." *ACM Transactions on Computer Systems*. Vol. 3, No. 4, pp. 261-293: 1985.
- [BGMR96] Bharadwaj, V., D. Ghose, V. Mani, and T. Robertazzi. *Scheduling Divisible Loads in Parallel and distributed Systems*. IEEE Computer Society Press, Los Alamitos, California, 1996.
- [Bla90] Black, D. "Scheduling Support for Concurrency and Parallelism in the Mach Operating system." *IEEE Computer*, Vol 23, No. 5, pp. 35-43:1990
- [Bla96] Black, U. *Mobile and Wireless Networks*. Prentice Hall, Englewood Cliffs, New Jersey, 1996.
- [Blo92] Bloomer, J. *Power Programming with RPC*. O'Reilly & Associates, Inc; Sebastopol, California, 1992.

- [BLPv93] Barnett, M., R. Littlefield, D. Payne, and R. van de Geijn. "Efficient Communication Primitives on Mesh Architectures with Hardware Routing." *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*. R. Sincovec, K. Keyes, M. Leuze, L. Petzold, and D. Reed (eds.) Vol. II, pp. 943-948: 1993.
- [Bra80] Braun, W. "Short Term Frequency Effects on Networks of Coupled Oscillators." *IEEE Transactions on Communications*. Vol. Com-28, No. 8, pp. 1269-1275: 1980.
- [Bra95] Braun, C. *UNIX System Security Essentials*. Addison Wesley; Reading, Massachusetts, 1995.
- [Bri75] Brinch Hansen, P., "The Programming Language Concurrent Pascal." *IEEE Transactions on Software Engineering*. Vol. SE-1, No. 6, pp. 199-207: 1975.
- [BrJo94] Brazier, F. and D. Johansen Eds. *Distributed Open Systems*. IEEE Press, Los Alamitos, California, 1994.
- [BSCEFHL93] Bauer, M., E. Strom, N. Coburn, D. Erickson (Galli), P. Finnigan, J. Hong, P. Larson, and J. Slonim. "Issues in Distributed Architectures: A Comparison of Two Paradigms." *Proceedings of the 1993 International Conference on Open Distributed Processing*. Berlin, Germany, pp. 411-418: 1993.
- [BuScSu97] Buttazzo, T. and S. Scuola Superiore. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Hingham, MA: 1997.
- [BZS93] Bershad, B., M. Zekauskas, and W. Sawdon. "The Midway Distributed Shared Memory System." *Proceedings of the IEEE COMP CON Conference*. pp. 528-537: 1993.
- [CAKLMS92] Chutani, S., O. Anderson, M. Kazar, B. Leverett, W. Mason, and R. Sidebothan. "The Episode File System." *Proceedings of the 1992 USENIX Winter Conference*. pp. 43-60: 1992.
- [CaKu88] Casavant, T. and J. Kuhl. "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems." *IEEE Transactions on Software Engineering*. Vol. 14, No. 2, pp. 141-154: 1988.
- [CaMu94] Casavant, T. and M. Singhal (Eds.). *Readings in Distributed Computing Systems*. IEEE Press, Los Alamitos, California, 1994.
- [CAPK98] Cabrera, L., B. Andrew, K. Peltonen, and N. Kusters. "Advanced in Windows NT Storage Management." *IEEE Computer*. Vol. 31, No. 10, pp. 48-54: 1998.
- [Car95] Carter, D. *Computer Crime in America*. Technical Report, Michigan State University, 1995.
- [CaRo83] Carvalho, O. and G. Roucairol. "On Mutual Exclusion in Computer Networks." *Communications of the ACM*. Vol. 26, No. 2, pp. 146-147: 1983.
- [CBE95] Chen, K., R. Bunt, and D. Eager. "Write Caching in Distributed File Systems." *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*. IEEE Press, New York, New York: 1995.
- [Cha97] Chan, T. *UNIX System Programming Using C++*. Prentice Hall PTR; Upper Saddle River, New Jersey, 1996.
- [ChBe94] Cheswick, W. and S. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley Publishing Co.; Reading, Massachusetts, 1994.
- [ChZw95] Chapman, D. and E. Zwicky. *Building Internet Firewalls*. O'Reilly & Associates, Inc; Sebastopol, California, 1995.

- [CIRM93] Campbell, R., N. Islam, D. Raila, and P. Madany. "Designing and Implementing Choices: An Object-Oriented System in C++." *Communications of the ACM*. Vol. 36, No. 9, pp. 117-126: 1993.
- [Cla85] Clark, D. "The Structuring of Upcalls." *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*. Pp. 171-180: 1985.
- [CIHo94] Clark, P. and L. Hoffman. "BITS: A Smartcard protected Operating System." *Communications of the ACM*. Vol. 37, No. 11, pp. 66-70: 1994.
- [Cou98] Courtois, T. *JAVA: Networking & Communications*. Prentice Hall, Englewood Cliffs, New Jersey: 1998.
- [Cri89] Cristian, F. "Probabilistic Clock Synchronization." *Distributed Computing*. Vol 3, pp. 146-158: 1989.
- [Cve87] Cvetanovic, Z. "The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems." *IEEE Transactions on Computers*. Vol. C-36, No. 4, pp. 421-432: 1987.
- [DaBu85] Davcev, D. and W. Burkhard. "Consistency and Recovery Control for Replicated Files." *Proceedings of the 10th ACM Annual Symposium on Operating Systems Principles*. pp. 87-96: 1985.
- [DDK94] Coulouris, G., J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design Second Edition*. Addison Wesley, Reading, Massachusetts, 1994.
- [DeDe94] Deitel, H. and P. Deitel. *C++ How to Program*. Prentice Hall, Englewood Cliffs, New Jersey: 1994.
- [DeDe98] Deitel, H. and P. Deitel. *Java How to Program*. Prentice Hall, Englewood Cliffs, New Jersey: 1998.
- [DeSa81] Denning, D. and G. Sacco. "Timestamps in Key Distribution Protocols." *Communications of the ACM*. Vol. 24, No. 8, pp. 533-536: 1981.
- [DH76] Diffie, W. and M. Hellman. "New Directions in Cryptography." *IEEE Transactions on Information Theory*. Vol. IT-11, No. 11, pp. 644-654: 1976.
- [Dij65] Dijkstra, E. "Co-operating Sequential Processes." *Programming Languages*. F. Genuys (Ed.) Academic Press, London: 1965.
- [Dio80] Dion, J. "The Cambridge File Server." *ACM Operating Systems Review*. Vol. 14, No. 4, pp. 26-35: 1980.
- [DLAR91] Dasgupta, P., R. LeBlanc Jr., M. Ahamad, and U. Ramachandran. "The Clouds Distributed Operating System." *IEEE Computer*. Vol. 24, No. 11, pp. 34-44: 1991.
- [DoMc98] Dowd, P. and J. McHenry. "Network Security: It's Time to Take it Seriously." *IEEE Computer*. Vol. 31, No. 9, pp. 24-28: 1998.
- [DSB86] Dubois, M., C. Scheurich, and F. Briggs. "Memory Access Buffering in Multiprocessors." *Proceedings of the 13th ACM Annual International Symposium on Computer Architecture*. Pp. 434-442: 1986.
- [DSB88] Dubois, M., C. Scheurich, and F. Briggs. "Synchronization, Coherence, and Event Ordering in Multiprocessors." *IEEE Computer*. Vol. 21, No. 2, pp. 9-21: 1988.
- [ErCo93a] Erickson (Galli), D. and C. Colbourn, *Combinatorics and the Conflict-Free Access Problem*, Congressus Numerantium, Vol. 94, pp. 115-121, December 1993.
- [ErCo93b] Erickson (Galli), D. and C. Colbourn, *Conflict-Free Access for Collections of Templates*, Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, Vol. II, pp. 949-952, March 1993.
- [Erick93a] Erickson (Galli), D. *Threshold Schemes with Hierarchical Information*,

- Technical Report #CS-93-23, University of Waterloo, Department of Computer Science, ftp only. (<ftp://cs-archive.uwaterloo.ca/cs-archive/CS-93-23/23Th.Z>) 81 pages, 1993.
- [Erick93b] Erickson (Galli), D. *Conflict-Free Access to Rectangular Subarrays in Parallel Memory Modules*, Technical Report #CS-93-24, University of Waterloo, Department of Computer Science, ftp only. (<ftp://cs-archive.uwaterloo.ca/cs-archive/CS-93-24/Thesis.Z>) 102 pages, 1993.
- [ErCo92] Erickson (Galli), D. and C. Colbourn, *Conflict-Free Access to Rectangular Subarrays*, *Congressus Numerantium*, Vol. 90, pp. 239-253, November 1992.
- [ELZ86] Eager, D., E. Lazowska, and J. Zahorjan. "Adaptive Load Sharing in Homogeneous Distributed Systems." *IEEE Transactions on Software Engineering*. Vol. SE-12, No. 5, pp. 662-675: 1986.
- [EAL95] El-Rweini, H., H. Ali, and T. Lewis. "Task Scheduling in Multiprocessing Systems." *IEEE Computer*. Vol. 28, No. 12, pp. 27-37: 1995.
- [ELZ86] Eager, D., E. Lazowska, and J. Zahorjan. "Adaptive Load Sharing in Homogeneous Distributed Systems." *IEEE Transactions on Software Engineering*. Vol. SE-12, No. 5, pp. 662-675: 1986.
- [Esk89] Eskicioglu, M. "Design Issues of Process Migration Facilities in Distributed Systems." *IEEE Technical Committee on Operating Systems Newsletter*. Vol. 4, No. 2, pp. 3-13: 1989.
- [Esk95] Eskicioglu, M., B. Shirazi, A. Hurson, and K. Kavi (Eds.) "Design Issues of Process Migration Facilities in Distributed Systems." *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, California, pp. 414-424: 1995.
- [Fer89] Fernandez-Baca, D. "Allocating Modules to Processors in a Distributed System." *IEEE Transactions on Software Engineering*. Vol. 15, No. 11, pp. 1427-1436: 1989.
- [Fer95] Ferrari, D. "A New Admission Control Method for Real-Time Communication in an Internetwork." *Advances in Real-Time Systems*. S. Son (ed.) Prentice Hall PTR; Upper Saddle River, New Jersey, 1995.
- [FeZh87] Ferrari, D. and S. Zhou. "An Empirical Investigation of Load Indices for Load Balancing Applications." *Proceedings Performance '87, The 12th Annual International Symposium on Computer Performance Modeling, Measurement and Evaluation*. North Holland Publishers, The Netherlands. Pp. 515-528: 1987.
- [Fla96] Flanagan, D. *Java in a Nutshell*. O'Reilly & Associates, Inc.; Sebastopol, California, 1996.
- [FrOl85] Fridrich, M. and W. Older. "Helix: the Architecture of the XMS Distributed File System." *IEEE Computer*. Vol. 18, No. 1, pp. 21-29: 1985.
- [Fer89] Fernandez-Baca, D. "Allocating Modules to Processors in a Distributed System." *IEEE Transactions on Software Engineering*. Vol. 15, No. 11, pp. 1427-1436: 1989.
- [Fu97] Fu, S. "A Comparison of Mutual Exclusion Algorithms for Distributed Memory Systems." *IEEE Newsletter of the Technical Committee on Distributed Processing*. pp. 15-20, Summer 1997.
- [FYN88] Ferguson, D., Y. Yemini, and C. Nikolaou. "Microeconomic Algorithms for Load Balancing in Distributed Computer Systems." *Proceedings of the Eight IEEE International Conference on Distributed Computing Systems*. pp. 491-499: 1988.
- [GaCo95] Galli, D. and C. Colbourn, *Conflict-Free Access to Constant-Perimeter Rectangular Subarrays* (Book Chapter), edited by F. Hsu, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 21, pp. 105-124, 1995.

- [Gar82] Garcia-Molina, H. "Elections in a Distributed computing System." *IEEE Transactions on Computers*. Vol. 31, No. 1, pp. 48-59: 1982.
- [GaSp91] Garcia-Molina, H. and A. Spauster. "Ordered and Reliable Multicast Communication." *ACM Transactions on Computer Systems*. Vol. 9, No. 3, pp. 242-271: 1991.
- [GaShMa98] Gamache, R., R. Short, and M. Massa. "The Windows NT Clustering Service." *IEEE Computer*. Vol. 31, No. 10, pp. 55-62: 1998.
- [GaSp96] Garfinkel, S. and G. Spafford. *Practical UNIX & Internet Security*. O'Reilly & Associates, Inc., Cambridge, 1996.
- [Gen81] Gentleman, W. M. "Message Passing Between Sequential Processes: The Reply Primitive and the Administrator Concept." *Software-Practice and Experience*. Vol. II, pp. 435-466: 1981.
- [GeYa93] Gerasoulis, A. and T. Yang. "On the Granularity and Clustering of Directed Acyclic Task Graphs." *IEEE Transactions on parallel and Distributed Systems*. Vol. 4, No. 6, pp. 686-701: 1993.
- [GGH91] Gharachorloo, K., A. Gupta, and J. Hennessy. "Performance Evaluations of Memory Consistency Models for Shared-Memory Multiprocessors." *Proceedings of the 4th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*. Pp. 245-257: 1991.
- [GhSc93] Gheith, A. and K. Schwan. "Chaos[™]: Kernel Support FOR Multi-WEIGHT Objects, Invocations and Atomicity in Real-Time Multiprocessor Applications." *ACM Transactions on Computer Systems*. Vol. 11, No. 1, pp. 33-72: 1993.
- [Gib87] Gibbons, P. "A Stub Generator for Multi-Language RPC in Heterogeneous Environment." *IEEE Transactions on Software Engineering*. Vol. SE-13, No. 1, pp. 77-87: 1987.
- [GiGl88] Gifford, D. and N. Glasser. "Remote Pipes and Procedures for Efficient Distributed Communication." *ACM Transactions on Computer Systems*. Vol. 6, No. 3, pp. 258-283: 1988.
- [GLLGGH90] Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. "Memory Consistency and Event Ordering in Scalable Shared-Memory Processors." *Proceedings of the 17th ACM Annual International Symposium on Computer Architecture*. pp. 15-26: 1990.
- [GNS88] Gifford, D., R. Needham, and M. Schroeder. "The Cedar File System." *Communications of the ACM*. Vol. 31, No. 3, pp. 288-298: 1988.
- [Goo89] Goodman, J. *Cache Consistency and Sequential Consistency*. Technical Report No. 61, IEEE Scalable Coherent Interface Working Group, IEEE, New York: 1989.
- [Gra78] Gray, J. "Notes on Database Operating Systems." R. Bayer, R. Graham, and G. Seegmuller (eds.) *Operating Systems: An Advance Course*. Springer-Verlag, Berlin, Germany: pp. 394-481: 1978.
- [Gra97] Gray, J. *Interprocess Communications in UNIX: The Nooks & Crannies*. Prentice Hall PTR, Upper Saddle River, New Jersey, 1996.
- [GrRe93] Gray, J., and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, California: 1993.
- [GuZa89] Gusella, R. and S. Zatti. "The Accuracy of the Clock synchronization Achieved by TEMPO in Berkely UNIX 4.3 BSD." *IEEE Transactions on Software Engineering*. Vol. SE-15, No. 7, pp. 847-853: 1989.
- [Hal96] Halsall, F. *Data Communications, Computer Networks, and Open Systems*. Addison Wesley, Reading Massachusetts, 1996.

- [HaOs93] Harrison, W. and H. Ossher, *Subject-Oriented Programming (A Critique of Pure Objects)*, Proceedings of the conference on object-oriented programming: Systems, languages, and applications, Washington, D.C., ACM: pp. 411-428, 1993.
- [HaRe83] Harder, T. and A. Teuter. "Principles of Transaction-Oriented Database Recover." *ACM Computing Surveys*. Vol. 15, No. 4: 1983.
- [HaSe98] Haggerty, P. and K. Seetharaman. "The Benefits of CORBA-Based Network Management." *Communications of the ACM*. Vol. 41, No. 10, pp: 73-79: 1998.
- [Hel78] Hellman, M. "An Overview of Public-Key Cryptography." *IEEE Transactions on Computers*. Vol. C-16, No. 6, pp. 24-31: 1978.
- [Hen98] Henning, M. "Binding, Migration, and Scalability in CORBA." *Communications of the ACM*. Vol. 41, No. 10, pp: 62-72: 1998.
- [Her94] Herbert, A. "Distributing Objects." Brazier and Johansen (Eds.) *Distributed Open Systems*, IEEE Computer Society Press, Los Alamitos, California, pp. 123-133: 1994.
- [HKMNSSW88] Howard, H., M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. "Scale and Performance in a Distributed File System." *ACM Transactions on Computer Systems*. Vol. 6, No. 1, pp. 51081: 1988.
- [Hoa74] Hoare, C. A. R. "Monitors, An Operating System Structuring Concept." *Communications of the ACM*. Vol. 17, No 10, pp. 549-557: 1974.
- [HSSD84] Halpern, J., B. Simons, R. Strong, and D. Dolly. "Fault-tolerant Clock Synchronization." *Proceedings of the Third Annual ACM Symposium on Distributed Systems*. pp. 89-102: 1984.
- [Hu95] Hunter, P. *Network Operating Systems: Making the Right Choices*. Addison Wesley, Reading, Massachusetts, 1994.
- [HuAh90] Hutto, P. and M. Ahamad. "Low Memory: Weakening Consistency to Enhance Cponcurrency in distributed Shared Memories." *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems*. Pp. 302-311: 1990.
- [HuPe91] Hutchinson, N. and L. Peterson. "The X-Kernel: An Architecture for Implementing Network Protocols." *IEEE Transactions on Software Engineering*. Vol. 17, No.1, pp. 64-76: 1991.
- [ISL96] Iftode, L., J. Pal Singh, and K. Li. "Scope Consistency: A Bridge between Release Consistency and Entry Consistency." *Proceedings of the 8th Annual Symposium on Parallel Algorithms and Architecture*. pp. 277-287: 1996.
- [IEEE85a] Institute of Electrical and Electronics Engineers. *802.3 CSMA/CD Access Method and Physical Layer Specifications*. IEEE, Standard 802.3, 1985.
- [IEEE85b] Institute of Electrical and Electronics Engineers. *802.5 Token Ring Access Method and Physical Layer Specifications*. IEEE, Standard 802.5, 1985.
- [ISO84] International Standards Organization. "Open System Interconnection". ISO Number 7498: 1984.
- [Jal94] Jalote, P. *Fault Tolerance in Distributed Systems*. Prentice Hall, Englewood Cliffs, New Jersey: 1994.
- [Joh95] Johnson, T. "A Performance Comparison of Fast Distributed Mutual Exclusion Algorithms." *Proceedings of the 9th International Parallel Processing Symposium*. Pp. 258-264: April 1995.
- [JoHa97] Johnson, T. and K. Harathi. "A Prioritized Multiprocessor Spin Lock." *IEEE Transactions on Paralle and Distributed Systems*. Vol. 8, No. 9. Pp. 926-933: 1997.
- [KCZ92] Keleher, P., A. Cox, W. Zwaenepoel. "Lazy Release Consistency." *Proceedings*

of the 19th ACM International Symposium on Computer Architecture. pp. 13-21: 1992.

[KeLi89] Kessler, R. and M. Livney. "An Analysis of Distributed Shared Memory Algorithms." *Proceedings Ninth International Conference on Distributed Computing Systems*. CS Press, pp. 498-505: 1989.

[Kim97] Kim, K. H. "Object Structures for Real-Time Systems and Simulators." *IEEE Computer*. Vol. 30, No. 8, pp. 62-70: 1997.

[KiPu95] Kim, T. and J. Purtilo. "Configuration-Level Optimization of RPC-Based Distributed Programs." *Proceedings of the 15th International Conference on distributed Computing Systems*. IEEE Press, Piscataway, New Jersey: 1995.

[KJAKL93] Kranz, D., K. Johnson, A. Agarwal, J. Kubiawicz, and B. Lim. "Integrating Message Passing and Shared Memory: Early Experiences." *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming (ACM)*. pp. 54-63: 1993.

[Klu94] Kluepfel, H. "Securing a Global village and Its Resources." *IEEE Communications Magazine*. pp. 82-89: September 1994.

[Kna87] Knapp, E., "Deadlock Detection in Distributed Databases." *ACM Computing Surveys*. Vol. 19, No. 4, pp. 303-328: 1987.

[KoOc87] Kopertz, H. and W. Ochsenreiter. "Clock Synchronization in Distributed Real-time Systems." *IEEE Transactions on Computers*. Vol. C-36, No. 8, pp. 933-939: 1987.

[Kop97] Kopetz, H. *Real-Time Systems Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Hingham, MA: 1997.

[KPS95] Kaufman, C., R. Perlman, and M. Speciner. *Network Security: PRIVATE Communication in a PUBLIC World*. Prentice Hall, Upper Saddle River, New Jersey: 1995.

[Kri89] Krishnamurthy, E. *Parallel Processing: Principles and Practice*. Addison Wesley, Reading Massachusetts, 1989.

[KrLi87] Krueger, P. and M. Livney. "The Diverse Objectives of Distributed Scheduling Policies." *Proceedings IEEE 7th International Conference on Distributed Computing Systems*. Pp. 242-249: 1987.

[KSS97] Kleiman, S., D. Shah, and B. Smaalders. *Programming with Threads*. SunSoft Press: A Prentice Hall Title: Sun Microsystems, Mountain View California: 1997.

[Kum91] Kumar, A., "Hierarchical Quorum Consensus: A New Algorithms for Managing Replicated Data." *IEEE Transactions on Computers*. Vol. 40, No. 9, pp. 996-1004: 1991.

[Kun91] Kunz, T. "The Influence of Different Workload Descriptions on Heuristic Load Balancing Scheme." *IEEE Transactions on Software Engineering*. Vol. 17, No. 7, pp. 725-730: 1991.

[LABW94] Lampson, B., M. Abadi, M. Burrows, and W. Wobber. "Authentication in distributed Systems: Theory and Practice." *ACM Transactions on Computer Systems*. Vol. 10, No. 4, pp. 265-310: 1992.

[LaEl91] LaRowe, R. and C. Ellis. "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors." *ACM Transactions on Computer Systems*. Vol. 9, pp. 319-363: 1991.

[Lai92] Lai, X. *On the Design and Security of Block Ciphers: Series in Information Processing*. Hartung-Corre Verlag, Konstanz, Germany: 1992.

- [Lam78] Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM*. Vol. 8, No. 7, pp. 558-564: 1978.
- [Lam79] Lamport, L. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs." *IEEE Transactions on Computers*. Vol. C-28, No. 9, pp. 690-691: 1979.
- [Lam81] Lamport, B. "Atomic Transactions in Distributed Systems-Architecture and Implementation." *An Advanced Course, Lecture Notes in Computer Science Vol. 105*. B. Lampson, M. Paul and H. Siegart (Eds.) Springer-Verlag, New York, pp. 246-264.
- [LaMe85] Lamport, L. and P. Melliar-Smith. "Clock Synchronization in the Presence of Faults." *Journal of the ACM*. Vol. 32, No. 1, pp. 52-78: 1985.
- [Lap92] Laplante, P. *Real-Time Systems Design and Analysis: An Engineer's Handbook*. IEEE Press, Piscataway, New Jersey: 1992.
- [Lap97] Laplante, P. *Real-Time Systems Design and Analysis: An Engineer's Handbook, Second Edition*. IEEE Press, Piscataway, New Jersey: 1997.
- [LeBe96] Lewis, B. and D. Eerg. *THREADS PRIMER: A guide to multithreaded Programming*. SunSoft Press: A Prentice Hall Title: Sun Microsystems, Mountain View California: 1997.
- [Lee97] Lee, P. "Efficient Algorithms for Data Distribution on Distributed Memory Parallel Computers." *IEEE Transactions on Parallel and Distributed Systems*. Vol 8, No. 8, pp. 825-839: 1997.
- [LEK91] LaRowe, R., C. Ellis, and L. Kaplan. "The Robustness of NUMA Memory Management." *Proceedings of the 13th ACM Symposium on Operating System Principles*. pp. 137-151: 1991.
- [LeSi90] Levy, E. and A. Silberschatz. "Distributed File Systems: Concepts and Examples." *ACM Computing Surveys*. Vol. 22, No. 4, pp. 321-374: 1990.
- [LHWS96] Loh, P. K. K., W. J. Hsu, C. Wentong, and N. Sriskanthan. "How Network Topology Affects Dynamic Load Balancing." *IEEE Parallel & Distributed Technology: Systems and Applications*. Vol. 4, No. 3, pp. 25-35: 1996.
- [LiHu89] Li, K. and P. Hudak. "Memory Coherence in Shared Virtual Memory Systems." *ACM Transactions on Computer Systems*. Vol. 7, pp. 321-359: 1989.
- Systems." *ACM Transactions on Computer Systems*. Vol. 7, pp. 321-359: 1989.
- [LiKa80] Lindsay, W. and A. Kantak. "Network Synchronization of Random Signals." *IEEE Transactions on Communications*. Vol. Com-28, No. 8, pp. 1260-1266: 1980.
- [Lin95] Lin, K. J. "Issues on Real-Time Systems Programming: Language, Compiler, and Object Orientation." In Son, S. H. (Ed.) *Advances in Real-Time Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.
- [LiSa88] Lipton, R. and J. Sandberg. *PRAM: A Scalable Shared Memory*. Princeton University Computer Science Technical Report: CS-TR-180-88: 1988.
- [LJP93] Lea, R., C. Jacquernot, and E. Pillevesse. "COOL: System Support for Distributed Programming." *Communications of the ACM*. Vol. 36, No. 9, pp. 37-46: 1993.
- [LoKi97] Lopes, C. and D. Kiczales. *A Language For Distributed Programming*, Xerox Palo Alto Research Center, Palo Alto, CA, 1997.
- [Lo88] Lo, V. M. "Heuristic Algorithms for Task Assignment in Distributed Systems." *IEEE Transactions on Computers*. Vol. 37, No. 11, pp. 1384-1397: 1988.
- [LuLy84] Lundelius, J. and N. Lynch. "A New Fault Tolerant Algorithm for Clock Synchronization." *Proceedings of the Third Annual ACM Symposium on Principles of*

Distributed Computing. Pp. 75-88: 1984.

[LZCZ86] Lazowska, E., J. Zahorjan, D. Cheriton, and W. Zwaenepoel. "File Access Performance of Diskless Workstations." *ACM Transactions on Computer Systems*. Vol. 4, No. 3, pp. 238-268: 1986.

[MaOw85] Marzullo, K., and S. Owicki. "Maintaining Time in a Distributed System." *ACM Operating Systems Review*. Vol. 19, No. 3, pp. 44-54: 1985.

[MaSm88] Maguire, G. Jr., and J. Smith. "Process Migration: Effects on Scientific Computation." *ACM SIGPLAN Notices*. Vol. 23, No. 3, pp. 102-106: 1988.

[Makk94] Makki, K. "An Efficient Token-based Distributed Mutual Exclusion Algorithm." *Journal of Computer and Software Engineering*. December 1994.

[McC98] McCarthy, L. *Intranet Security: Stories from the Trenches*. Prentice Hall, Upper Saddle River, New Jersey: 1998.

[MDGM99] McReynolds, D., S. Duggins, D. Galli, and J. Mayer. "Distributed Characteristics of Subject Oriented Programming: An Evaluation with the Process and Object Paradigms." *Proceedings of the ACM SouthEastern Regional Conference*, ACM, Mobile Alabama: to appear 1999.

[Micr98] Microsoft Corporation. *NT White Paper Series*. www.microsoft.com/ntserver, accessed 1998.

[MiDi82] Mitchell, J. and J. Dion. "A Comparison of Two Network Based File Servers." *Communications of the ACM*. Vol. 25, No. 4, pp. 233-245: 1982.

[Mil88] Mills, D. "The Fuzzball." *Proceedings of ACM's SIGCOMM Symposium*. Pp. 115-122: 1988.

[Mil90] Mills, D. "Measured Performance of the Network Time Protocol in the Internet System." *ACM Computer Communication Review*. Vol. 20, No. 1, pp. 65-75: 1990.

[Mil91a] Mills, D. "Internet Time Synchronization: The Network Time Protocol." *IEEE Transactions on Communications*. Vol. 39, No. 10, pp. 1482-1493: 1991.

[Mil91b] Mills, D. "On the Chronology and Metrology of Computer Network Timescales and their Application to the Network Time Protocol." *ACM Computer Communications Review*. Vol. 21, No. 5, pp. 8-17: 1991.

[Mil92] Mills, D. "Network Time Protocol Version 3: Specification, Implementation and Analysis." *Network Working Group Request For Comments*. No. 1305: March, 1992.

[Mil95] Mills, D. "Simple Network Time Protocol (SMTP)." *Network Working Group Request For Comments*. No. 1769: March, 1995.

[Mit80] Mitra, D. "Network Synchronization: Analysis of a Hybrid of Master/Slave Synchronization." *IEEE Transactions on Communications*. Vol. Com-28, No. 8, pp. 1245-1258: 1980.

[MoPu97] Morin, C. and I. Puaut. "A Survey of Recoverable Distributed Shared Virtual Memory Systems." *IEEE Transactions on Parallel and Distributed Systems*. Vol. 8, No. 9, pp. 959-969: 1997.

[MSCHRS86] Morris, J., M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith. "Andrew: A Distributed Personal Computing Environment." *Communications of the ACM*. Vol. 29, No. 3, pp. 184-201: 1986.

[Mul93a] Mullender, S. (Ed.). *Distributed Systems: Second Edition*. ACM Press, New York, New York, 1993.

[Mul93b] Mullender, S. "Kernel Support for Distributed Systems." *Distributed Systems*:

Second Edition. ACM Press, New York, New York, pp. 385-410: 1993.

[MuLi87] Mutka, M. and M. Livney. "Scheduling Remote Processor Capacity in a Workstation-Processor Bank Network." *Proceedings of the Seventh IEEE International Conference on Distributed Computing Systems*. Pp. 2-9: 1987.

[MVTVV90] S. Mullender, G. VanRossum, A. Tanenbaum, R. VanRenesse, and H. VanStaveren. "Amoeba: A Distributed Operating System for the 1990's," *IEEE Computer*, Vol 23, No. 5, pp. 44-53:1990.

[NeBi84] Nelson, B. and A. Birrell. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems*. Vol. 1, No. 2, pp. 39-59: 1984.

[NeTs94] Neuman, C. and T. Ts'O. "Kerberos: An Authentication Service for Computer Networks." *IEEE Communications*, Vol. 32, No. 9, pp. 33-38: 1994.

[NCSC94] National Computer Security Center, *Final Evaluation Report: Cray Research Incorporated Trusted UNICOS 8.0*. CSC-EPL-94/002, C-Evaluation No. 27/94. Fort George G. Meade, Maryland, March, 1994.

[Nel81] Nelson, B. *Remote Procedure Call*. Ph.D. Thesis, Carnegie-Mellon University: 1981.

[Nes83] Nessett, D. "A Systematic methodology for analyzing Security Threats to Interprocess Communications in a Distributed System." *IEEE Transactions on Communications*. Vol. COM-31, pp. 1065-1063: 1983.

[Nev95] Nevison, C. "Parallel Computing in the Undergraduate Curriculum." *IEEE Computer*. Vol. 28, No. 12: pp. 51-56: 1995.

[NiPe96] Niemeyer, P. and J. Peck. *Exploring Java*. O'Reilly & Associates, Inc.; Sebastopol, California, 1996.

[NIST77] NIST. *Data Encryption Standard*. Federal Information Processing Standards publication 46, January 15, 1977.

[NIST93a] NIST (J. Barkley). *Comparing Remote Procedure Calls*. National Institute for Standards and Technology Information Report 5277: 1993.

[NIST93b] NIST. *Secure Hash Standard (SHS)*. Federal Information Processing Standards publication 180: 1993.

[NIST94a] NIST. *Escrowed Encryption Standard*. Federal Information Processing Standards publication 185: 1994.

[NIST94b] NIST. *Digital Signature Standard (DSS)*. Federal Information Processing Standards publication 186: 1994.

[NWO88] Nelson, M., B. Welch, and J. Ousterhout. "Caching in the Sprite Network File System." *ACM Transactions on Computer Systems*. Vol. 6, No. 1, pp. 134-154: 1988.

[OMG97] Object Management Group. *The Common Object Request Broker: Architecture and Specification: Version 2.1*. OMG, Framingham, Massachusetts, August 1997.

[OMG98a] Object Management Group. *The Common Object Request Broker: Architecture and Specification: Version 2.2*. OMG, Framingham, Massachusetts, February 1998.

[OMG98b] Object Management Group. *CORBAtelecoms: Telecommunications Domain Specifications: Version.1*. OMG, Framingham, Massachusetts, June 1998.

[Opp97] Opplinger, R. "Internet Security: Firewalls and Beyond." *Communications of the ACM*. Vol. 40, No. 5, pp. 92-102: 1997.

[Opp98] Opplinger, R. "Security at the Internet Layer." *IEEE Computer*. Vol. 31, No. 9, pp.

43-47: 1998.

[OsHa95] Ossher, H. and W. ????, *Subject-Oriented Programming: Supporting Decentralized Development of Objects*, Research Report RC 20004, IBM Thomas J. Watson Research Center, Yorktown Heights, NY March 1995.

[Ous82] Ousterhout, J. "Scheduling Techniques for Concurrent systems" *Proceedings of the Third International Conference on Distributed Computing Systems*: Ft. Lauderdale, FL, IEEE Computer Science Press, pp: 22-30.

[PaSh88] Panzieri, F. and S. Shrivastava. "Rajdoot: A Remote Procedure Call Mechanism with Orphan Detection and Killing." *IEEE Transactions on Software Engineering*. Vol. 14, No. 1, pp. 30-37: 1988.

[PaSh98] Patiyoote, D., and S. Shepherd. "Techniques for Authentication Protocols and Key Distribution on Wireless ATM Networks." *Operating Systems Review*. ACM SIGOP Publication. Vol. 32, No. 4, pp. 25-32: 1998.

[Pf97] Pfleeger, C. *Security in Computing: Second Edition*. Prentice Hall, Upper Saddle River, New Jersey: 1997.

[Pow95] Power, R. *Current and Future Danger: A CSI Primer on Computer Crime and Information Warfare?* Computer Security Institute, 1995.

[Prim95] Primates, F. *Tuxedo: An Open Approach to OLTP, 1/e*. Prentice Hall, Upper Saddle River, New Jersey: 1995.

[PTM96] Protic, J., M. Tomasevic, and V. Milutinovic. "Distributed Shared Memory: Concepts and Systems." *IEEE Parallel & Distributed Technology: Systems & Applications*. Vol. 4, No. 2, pp. 63-79: 1996.

[PTM98] Protic, J., M. Tomasevic, and V. Milutinovic. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society Press, Los Alamitos, California: 1998.

[PTS88] Pulidas, S., D. Towsley, and J. Stankovic. "Imbedding Gradient Estimators in Load Balancing Algorithms." *Proceedings IEEE 8th International Conference on Distributed Systems*. Pp. 482-490: 1988.

[QuSh95] Quinn, B. and D. Shute. *Windows Sockets Network Programming*. Addison Wesley, Reading, Massachusetts, 1995.

[RaCh96] Ramamritham, K. and P. Chrysanthos (Eds.) *Advances in Concurrency Control and Transaction Processing*, IEEE Computer Society Press, Los Alamitos, California: 1996.

[RaSh92] Ramanathan, P. and K. Shin. "Delivery of Time-Critical Messages Using a Multiple Copy Approach." *ACM Transactions on Computer Systems*. Vol. 10, pp. 144-166: 1992.

[RiAg81] Ricard, G. and A. Agrawala. "An Optimal Algorithms for Mutual Exclusion in Computer Networks." *Communications of the ACM*. Vol. 34, No. 2, pp. 9-17: 1981.

[Ric88] Rickert, N. "Non-Byzantine Clock Synchronization: A Programming Experiment." *ACM Operating Systems Review*. Vol. 22, No. 1, pp. 73-78: 1988.

[RoRo96] Robbins, K. and S. Robbins. *Practical Unix Programming: A Guide to Concurrency, Communication, and Multithreading*. Prentice Hall PTR, Upper Saddle River, New Jersey, 1996.

[RSA78] Rivest, R., A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." *Communications of the ACM*. Vol. 21, No. 2, pp. 120-126: 1978.

[RuGe98] Rubin, A. and D. Geer Jr. "A Survey of Web Security." *IEEE Computer*.

Vol. 31, No. 9, pp. 34-41: 1998.

[San87] Sandberg, R. "The Sun Network File System: Design, Implementations and Experience." *Proceedings of the USENIX Summer Conference*. pp. 300-314: 1987.

[Sat90a] Satyanarayanan, M. "A Survey of Distributed File Systems." *Annual Review of Computer Science*. Vol. 4, pp. 73-104: 1990.

[Sat90b] Satyanarayanan, M. "Scalable, Secure, and Highly Available Distributed File System Design." *IEEE Transactions on Software Engineering*. Vol. 18, No. 1: 1992.

[ScBu89] Schroeder, M. and M. Burrows. "Performance of the Firefly RPC." *Twelfth ACM Symposium on Operating System Principles*. Pp. 83-90: 1989.

[Sch98] Schmidt, D. "Evaluating Architectures for Multithreaded Object Request Brokers." *Communications of the ACM*. Vol. 41, No. 10, pp: 54-61: 1998.

[Schn98] Schneier, B. "Cryptographic Design Vulnerabilities." *IEEE Computer*. Vol. 31, No. 9, pp. 29-33: 1998.

[ScPu94] Schwartz, M. and C. Pu. "Applying an Information Gathering Architecture to Netfind: A White Pages Tool for Changing and Growing Internet." *IEEE/ACM Transactions on Networking*. Vol. 2, No. 5, pp. 426-439: 1994.

[Sha77] Shankar, K. "The Total Computer Security Problem: An Overview." *IEEE Computer*. Vol. 10, pp. 50-62, 71-73: 1977.

[SHFECB93] Slonim, J., J. Hong, P. Finnigan, D. Erickson (Galli), N. Colburn, and M. Bauer. "Does Middleware Provide an Adequate Distributed Application Environment?" *Proceedings of the 1993 International Conference on Open Distributed Processing*. Berlin, Germany. Pp. 34-46: 1993.

[SHK95] Shirazi, B., A. Hurson, and K. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, California: 1995.

[ShSn88] Shasha, D. and M. Snir. "Efficient and Correct Execution of Parallel Programs that Share Memory." *ACM Transactions on Programming Languages and Systems*. Vol. 10, No. 2, pp. 282-312: 1988.

[Sie98] Siegel, J. "OMG Overview: CORBA and the OMA in Enterprise Computing." *Communications of the ACM*. Vol. 41, No. 10, pp: 37-43: 1998.

[Sim92] Simmons, G. (Ed). *Contemporary Cryptography*. IEEE, New York, 1992.

[Sin97] Sinha, P. *Distributed Operating Systems: Concepts and Design*. IEEE Computer Society Press, New York, 1977.

[SKS92] Shivaratri, N., P. Krueger, and M. Singhal. "Load Distribution for Locally Distributed Systems." *IEEE Computer*, Vol. 25, No. 12. pp. 33-44: 1992.

[SMI80] Sturgis, H., J. Mitchell, and J. Israel. "Issues in the Design and Use of a Distributed File System." *ACM Operating Systems Review*. Vol. 14, No. 3, pp. 55-69: 1980.

[SOL98] Solomon, D. A. "The Windows NT Kernel Architecture." *IEEE Computer*. Vol. 31, No. 10, pp. 40-47: 1998.

[Son95] Son, S.H. (Ed.) *Advances in Real-Time Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[SrPa82] Srivastava, S. and F. Panzieri. "The Design of Reliable Remote Procedure Call Mechanism." *IEEE Transactions on Computers*. Vol. C-31, No. 7: 1982.

- [SSF97] Shuey, R., D. Spooner, and O. Frieder. *The Architecture of Distributed Computing Systems*. Addison Wesley; Reading, Massachusetts, 1997.
- [SSRB98] Stankovic, J., M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, Hingham, MA, 1998.
- [Sta95] Stallings, W. *Network and Internetwork Security: Principles and Practice*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.
- [Sta97] Stallings, W. *Data and Computer Communications: Fifth Edition*. Prentice Hall, Englewood Cliffs, New Jersey, 1997.
- [Ste94] Stevens, W.R. *TCP/IP Illustrated, Vol. 1: The Protocols*. Addison Wesley, Reading Massachusetts, 1994.
- [StZh90] Stumm, M. and S. Zhou. "Algorithms Implementing Distributed Shared Memory." *IEEE Computer*. Vol. 23, No. 5, pp. 54-64: 1990.
- [Sun88] Sun Microsystems, Inc. *RPC: Remote Procedure Call: Protocol Specification Version 2*. Request for Comments 1057: 1988.
- [SuZu96] Sun, X.H. and J. Zhu. "Performance Prediction: A Case Study Using a Scalable Shared-Virtual-Memory Machine." *IEEE Parallel & Distributed Technology*. Vol. 4, No. 4, pp. 36-49: 1996.
- [SWP90] Shirazi, B., M. Wang, and G. Pathak. "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling." *Journal of Parallel and Distributed Computing*. Vol. 10, pp. 222-232: 1990.
- [TaAn90] Tay, F. and A. Ananda. "A Survey of Remote Procedure Calls." *Operating Systems Review*. Vol. 24, pp. 68-79: 1990.
- [TaKa94] Tanenbaum, A. and M. F. Kaashoek., Brazier and Johansen (Eds.) "The Amoeba Microkernel." *Distributed Open Systems*, IEEE Computer Society Press, Los Alamitos, California, pp. 11-30B: 1994.
- [Tan95] Tanenbaum, A. *Distributed Operating Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.
- [Tan96] Tanenbaum, A. *Computer Networks: Third Ed*. Prentice Hall, Englewood Cliffs, New Jersey, 1996.
- [Tho79] Thomas, R. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases." *ACM Transactions on Database Systems*. Vol. 4, No. 2, pp. 180-209: 1979.
- [THPSW98] Tarman, T., R. Hutchinson, L. Pierson, P. Sholander, and E. Witzke. "Algorithm-Agile Encryption in ATM Networks." *IEEE Computer*. Vol. 31, No. 9, pp. 57-64: 1998.
- [TLC85] Theimer, M., K. Lantz, and D. Cheriton. "Preemptive Remote Execution Facilities for the V System." *Proceedings of the 10th ACM Symposium on Operating System Principles*: 1985.
- [Vin98] Vinoski, S. "New Features for CORBA 3.0." *Communications of the ACM*. Vol. 41, No. 10, pp. 44-53: 1998.
- [VLL90] Veltman, B., B. Lageweg, and J. Lenstra. "Multiprocessor Scheduling with Communication Delays." *Parallel Computing*. Vol. 16, pp. 173-182: 1990.
- [WaMo85] Wang, Y. and R. Morris. "Load Sharing in Distributed Systems." *IEEE Transactions on Computers*. Vol. C-35, No. 3, pp. 204-217: 1985.
- [Wei91] Weihl, W. "Transaction-Processing Techniques." *Distributed Systems: 2nd Edition*. S. Mullender (Ed.). Association for Computing Machinery, New York, New York: pp. 329-352: 1993.

- [Wei93] Weikum, G., "Principles and Realization of Multilevel Transaction Management." *ACM Transactions on Database Systems*. Vol. 16, No. 1, pp. 132-140: 1991.
- [Wil94] Williams, D. *Authentication Protocols in Distributed Computer Networks*. Master's Thesis, Southern College of Technology: Dept. of Computer Science: Marietta, Georgia, 1994.
- [WoLa92] Woo, T. and S. Lam. "Authentication for Distributed Systems." *IEEE Computer*. Vol. 25, No. 1, pp. 39-52: 1992.
- [WuSa93] Wu, K. and Y. Saad. "Performance of the CM-5 and Message Passing Primitives." *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*. R. Sincovec, K. Keyes, M. Leuze, L. Petzold, and D. Reed (eds.) Vol. II, pp. 937-942: 1993.
- [YSL97] Yang, C., S. Shi, and F. Liu. "The Design and Implementation of a Reliable File Server." *IEEE Newsletter of the Technical Committee on Distributed Processing*. Summer 1997.
- [Zay87] Zayas, E. "Attacking the Process Migration Bottleneck." *Proceedings of the ACM Symposium on Operating Systems Principles*. Pp. 13-24: 1987.

索引

索引中的页码为英文原书的页码，与书中边栏的页码一致。

A

access control (访问控制), 259, 279

 firewall (防火墙), 279

access control list (访问控制表), 189

 in Windows 2000, 304

ACID property (原子性, 一致性, 独立性, 持久性), 207, 213, 228, 230, 231

 参见 transaction management

Active Directory (活动目录)

 参见 windows 2000

Active X, 138

adaptive assignment (自适应分派), 156

 based on economic usage point (基于经济使用观点), 159

 based on graph theory approach (基于图论方法), 160

 based on usage point (基于使用观点), 158

 centralized solution (集中式求解), 156

 decision permanence (决策持久性), 156

 distributed scheduler choice (分布式调度选择), 152

 distributed scheduling (分布式调度), 152

 distributed scheduling approach (分布式调度方法), 157

 distributed solution (分布式求解), 156

 efficiency goal (效率目标), 153

 efficiency metric (效率度量), 154

 global scheduling (全局调度), 153

 global scheduling algorithm type (全局调度算法类型), 157

 in Mach, 164

 level of scheduling (调度级别), 152

 load distribution goal (负载分布目标), 153

 mandatory scheduling (强制性调度), 156

 one-time assignment (一次性分派), 156

 optimal (最优), 153

 processor binding time (处理器捆绑时间), 155

 suboptimal (次优), 153

 suboptimal approximate solution (次优近似求解), 154

 suboptimal heuristic solution (次优启发式求解), 155

 using an omaton vector (使用 omaton 向量), 166

 using probe (使用监测器), 162

 using stochastic learning (使用随机的学习), 165

 utilizing queues (实用队列), 163

 voluntary scheduling (自愿调度), 156

 workload indicators (工作负荷指示器), 166

address space (地址空间)

 global (全局), 90

 in windows 2000, 298, 299

addressing (编址), 59

 explicit (显式), 59

 group (组), 60

 implicit (隐式), 59

 many-to-many (多对多), 61

 many-to-one (多对一), 60

 message ordering (消息排序), 62

 one-to-many (一对多), 60

 one-to-one (一对一), 59

Advanced Configuration and Power Interface (先进配置和电源界面)

参见 Windows 2000

Advanced Power Management (高级电源管理)

参见 Windows 2000

AFS Andrew (文件系统), 204

Amoeba, 135

memory management (内存管理)

参见 object communication (对象通信), 137

process management (进程管理), 40

ARPANET, 11

asymmetric cryptography (非对称加密), 137

asynchronous events (异构实体), 17

atomic action (原子操作), 107

authentication (可信度), 137, 246, 259, 270

certificate distribution center (认证分布中心), 275

certificate lists (认证表), 271

Kerberos, 276

X.509, 272

B

barrier (壁垒, 障碍), 222

busy wait (忙等待), 113

C

capabilities (能力), 189

casual consistency (偶然一致性), 217, 218

Chorus, 132

client/server model (C/S 模型), 12

COOL, 132

DSM, 91

memory management (内存管理), 96

client/server model (C/S 模型)

coordinator elections (对等选择), 167

in NTP, 247

Clouds, 131

cluster technology (集群技术), 290

in Windows 2000, 316

objects (对象), 131

threads (线程), 132

clusters (集群), 134

common external data representation (公共外部数据表示), 43

compound transaction (混合事务), 228

concurrency control (并发控制), 105

参见 deadlocks (死锁)

concurrent process scheduling (并发进程调度), 47

critical regions (临界区), 88, 106

distributed token manager (分布式令牌管理器), 135

hardware support (硬件支持), 113

in Clouds, 132

lock (锁), 111, 112

lock in Clouds (Clouds 中的锁), 132

mutex object in Windows 2000 (Windows 2000 中的互斥对象), 296

mutual exclusion (互斥), 105

race condition (竞争条件), 112

semaphore (信号量), 107

semaphore object in Windows 2000 (Windows 2000 中的信号量), 296

software lock (软件锁), 115

support in scheduling (调度支持), 165

synchronization (同步), 55

taking turn (轮转), 113

Three-point test (三点测试), 106

token-passing algorithm (令牌传递算法), 118

consistency model (一致性模型), 216

casual consistency (偶然一致性), 217, 218

entry release consistency (入口释放一致性), 224

lazy release consistency (懒惰释放一致性),

- 224
 - PRAM consistency (PRAM 一致性), 218, 219
 - processor consistency (处理器一致性), 219
 - release consistency (释放一致性), 222, 224
 - sequential consistency (顺序一致性), 216, 217, 221, 224
 - strict consistency (严格一致性), 216, 217
 - weak consistency (弱一致性), 220, 222, 224
 - consistency (一致性), 45, 88, 232
 - consistency model (一致性模型)
 - consistency in Windows 2000 (Windows 2000 中的一致性), 309
 - data consistency (数据一致性), 90
 - snoopy cache (窃听式高速缓存), 89
 - context switching (正文切换), 39
 - CORBA, 53, 127, 128, 132, 133, 143, 317
 - basic object adapter (基本对象适配器), 145
 - compliance (顺从, 遵循), 148
 - mapping to COM (映射到 COM), 148
 - messaging (消息传递), 147
 - object adapter (对象适配器), 145
 - ORB, 144
 - ORB in Chorus (Chorus 中的 ORB), 135
 - portable object adapter (灵活对象适配器), 145
 - critical region (临界区), 105, 108, 见 concurrency control
 - cryptography (加密)
 - RSA (三重公钥加密算法), 266
- D**
- Data Encryption Standard (数据加密标准), 197
 - 见 DES in NFS
 - datagram (数据报), 8, 67, 69
 - in Java, 70
 - DCOM, 53, 127, 138, 290
 - mapping from CORBA (从 CORBA 的映射), 148
 - deadlock (死锁), 119
 - avoid (避免), 121
 - condition (条件), 119
 - DES (数据加密标准), 261
 - detect (检测), 122
 - ignore (忽略), 121
 - PAID (处理死锁的方法), 120
 - prevent (防止), 120
 - phase 1 (阶段 1), 263
 - phase (阶段 2), 263
 - phase (阶段 3), 263
 - DES (数据加密标准)
 - algorithm (算法), 261
 - Diffie-Hellman key exchange (Diffie-Hellman 键变换), 269
 - in NTP, 246
 - in Windows (2000), 290
 - support for in Windows 2000 (对 Windows 2000 的支持), 319
 - digital signature (数字签名), 265
 - digest function (摘要功能), 265
 - message digest (消息摘要), 265
 - using public-key encryption (使用公共钥加密), 270
 - directory service (目录服务), 90, 97, 195
 - directory management (目录管理), 195
 - directory operations (目录操作), 196
 - directory structures (目录结构), 195
 - in Amoeba, 137
 - in NFS, 198
 - in Windows 2000, 290, 306
 - X.500, 200
 - distributed directory (分布式目录), 90
 - Active Directory (活动目录), 290

distributed file system (分布式文件系统)

见 file service

distributed real-time (分布式实时), 17

sample application (采样录用), 25

distributed scheduling (分布式调度)

见 scheduling

distributed shared memory (分布式共享内存), 89, 207, 222, 223, 231

multiple reader/multiple writer (多读者/多写者), 93

multiple reader/single writer (多读者/单写者), 91, 135

shared memory (共享内存)

single reader/single writer (单读者/单写者), 90

Domain Name Server (域名服务器), 184

in Windows 2000, 309

Domain Name System (域名系统)

use in Windows 2000, 306

domino effect (多米诺现象), 231

DSM

参见 distributed shared memory (分布式共享内存)

E

Encrypting File System (加密的文件系统)

参见 Windows 2000

encryption (加密), 260

asymmetric cryptography (非对称加密), 265

DES (数据加密标准), 261

digital signatures (数字签名), 265, 270

entry release consistency (入口(条目)释放一致性), 224

private-key (私钥), 261

public-key (公共钥), 265

symmetric encryption (对称加密), 261

synchronization variables (同步变量), 224

external data representation (外部数据表示),

43

fault tolerance (容错), 242

in NTP, 246

F

fault-tolerant (容错), 254

file replication (文件复制), 192

file service (文件服务), 187, 192

AFS (Andrew 文件系统), 204

file attribute (文件属性), 188

file modification notification (文件修改要点), 190

file protection mode (文件保护模式), 188

file replication (文件复制), 192

file service (文件服务), 207

file storage (文件存储), 187

file type (文件类型), 180

NTFS in Windows 2000 (Windows 2000 中的 NTFS), 303

reparse points in NTFS (NTFS 中的重解析点), 305

firewall (防火墙), 279

architecture (体系结构), 281

bastion host architecture (设防宿主体系结构), 281

filtering host architecture (过滤宿主体系结构), 282

filtering subnet architecture (过滤子网体系结构), 283

packet filtering (组过滤), 280

proxy server (代理服务器), 281

Flynn's taxonomy (Flynn 的分类), 21

G

global ordering (全局排序), 217, 235, 254

global time (全局时间), 216, 217, 235

drifting (偏移), 235

logical clock (逻辑时钟), 236

Physical clock (物理时钟), 236

H

happen-before relationship (超前关系), 250
 Hardware Abstraction Layer in Windows 2000 (Windows 2000 中的硬件抽象层)
 hash function (散列函数), 186
 heterogeneity (非均匀性, 多态性), 43
 HTTP 超文本传输协议
 support in Windows 2000, 311

I

interconnection network (互连网络)
 banyan, 23
 Benes, 23
 bus (总线), 20
 crossbar (交叉), 21
 hypercube (超立方体), 22
 interprocess communication (进程间通信), 53
 message passing (消息传输), 54
 pipe (管道, 流水线), 61
 remote procedure call (远程过程调用), 72
 selection factor (选择因子), 54
 ISO/OSI Reference Model (ISO/OSI 参考模型), 8, 14
 layer description (层次描述), 9
 point-to-point layer (点对点分层), 10
 isochronous event (等时事件), 17

J

Java, 128
 datagram (数据报), 70
 socket (套接字), 69
 Java,
 monitor (管程), 111
 multithreading support (多线程支持), 37
 JavaOS, 133

K

Kerberos, 197, 276, 230

support for in Windows 2000 (Windows 2000 中的支持), 319

kernel (内核), 31
 in Windows 2000 (Windows 2000 中), 293
 layered (分层), 32
 microkernel (微内核), 32
 monolithic (单一内核), 31
 kernel (内核), 81

M

micro-kernel (微内核), 133, 293, 320, 323, 324, 325
 参见 Windows 2000
 in Chorus, 134
 Windows 2000 kernel object (Windows 2000 内核对象), 295

L

Lamport's algorithm (莱氏算法), 62, 250, 254
 lazy release consistency (懒惰释放一致性), 224
 LDAP (轻量级目录存放协议)
 参见 active directory in Windows 2000, 289
 load balancing (负载平衡)
 参见 scheduling
 参见 migration
 load distribution (负载分布)
 参见 scheduling
 locks (封锁), 112
 参见 concurrency control
 centralized lock manager (集中式锁管理器), 115
 distributed lock manager (分布式锁管理器), 116
 happen-before relationship (超前关系), 250
 logical clock (逻辑锁), 236, 250
 logical ordering (逻辑排序), 251
 lost update (丢失恢复), 208, 213

M

Mach

scheduling (调度), 164

marshalling parameter (整理参数), 73

memory bottleneck (内存瓶颈), 87

memory management (内存管理), 81

参见 shared memory

参见 distributed shared memory

memory management unit (MMU) (内存管理单元 (MMU)), 82

in Chorus COOL, 134

in Clouds, 132

pages (页面), 82

page replacement algorithms (页面置换算法), 84

segmentation (分段, 碎片), 82

thrashing (颠簸), 85

memory migration (内存迁移)

concurrent copy (并发复制), 98

copy-on-reference (访问时复制), 98

freeze time (停顿时间), 98

stop-and-copy (停止和复制), 98

message ordering (消息排序), 62

incidental ordering (随机排序), 62

uniform ordering (均匀排序), 62

universal ordering (通用排序), 62

message passing (消息传递), 32, 54

blocking primitive (阻塞原语), 55

nonblocking (非阻塞), 55

nonblocking primitive (非阻塞原语), 58

message passing (消息传递), 86, 90

Microsoft Index Server (MS 索引服务器)

参见 Windows 2000

Microsoft Management Console (MS 管理控制台)

参见 Windows 2000

middleware (中间件), 15

migration (迁移), 129, 41

adaptive scheduling assignment (自适应调度

分派), 156

load balancing (负载均衡), 41, 153

load sharing (负载共享), 41, 153

migration (迁移), 88

memory migration (内存迁移), 98

support in Windows 2000 (Windows 2000 支持), 317

MIMD (多指令流多数据流), 21

MISD (多指令流单数据流), 21

monitors (管程), 108

参见 concurrency control

condition variable (条件变量), 110

support in Java (Java 中支持), 111

multicomputer (多计算机), 19

multiprocessors (多处理器), 19

multithreaded (多线程)

参见 thread

mutual exclusion (互斥)

参见 concurrency control

N

name service (名字服务), 179

in Clouds, 131

in Windows 2000, 308

in X.500, 201

name space (名字空间), 3

naming (命名), 131

Domain Name Server (域名服务器), 184

global naming (全局命名), 181

IPv4 naming (IPv4 命名), 184

nested transaction (嵌套事务), 228

NetWare (网件)

support for in Windows 2000 (Windows 2000 中支持), 304, 308

Network File System (网络文件系统), 197

directory service (目录服务), 198

mounting (装卸), 198

name service (名字服务), 199

network latency (网络延迟), 17

network system (网络系统), 3

Network Time Protocol (网络时间协议), 235, 243

- accuracy of (精确度), 254
- architecture (体系结构), 244
- authentication (授权), 246
- data format (数据格式), 246
- fault tolerance (容错), 246
- frame format (帧格式), 248
- Simple Network Time Protocol (简单网络时间协议), 247
- stratum (阶层), 244
- synchronization mode (同步模式), 247
- validation test (验证测试), 249

network, 4

- APPN, 14
- ATM, 8
- backbone (主干网), 4
- bridge (网桥), 4
- datagram (数据报), 8
- Ethernet (以太网), 5
- FDDI, 7
- frame relay (帧中继), 8
- interconnection network (互联网), 19
- LAN, 5
- PSTN (公共开关电话网络), 8
- repeaters (激励器), 4
- router (路由器), 4
- token ring (令牌环), 5, 7
- WAN, 8
- wireless (无线), 5, 7
- X.25, 14

NFS

参见 Network File System

NORMA architecture (NORMA 体系结构), 19

NTFS, 321

参见 Windows 2000

NUMA architecture (NUMA 体系结构), 19, 45, 82, 86

O

object (对象), 127, 128

- in Amoeba, 137
- in Clouds, 131
- in Chorus COOL, 133
- in Windows 2000 MMC, 313
- method (方法), 128
- multiple inheritance (多重继承), 128
- object adapter (对象捕获器), 145
- subject-oriented programming (面向主题程序设计), 130

Object Manager in Windows 2000 (Windows 2000 中的对象管理器), 298

- Windows 2000 kernel object (Windows 2000 内核对象), 295

orphan process handling (孤儿进程处理), 168

- child process allowance (子进程修正), 171
- orphan cleanup (孤儿清除), 170
- page replacement algorithm (页面替换淘汰算法), 84
- process version numbers (进程版本号), 173

P

page scanner (页面扫描器), 88

PAID (处理死锁的方法), 120

parallel software paradigms (并行软件范例), 23

parallel system (并行系统), 19

peer-to-peer model (同层模式), 13

- in NTP, 247

physical clocks (物理时钟), 236

- centralized physical time service (集中式物理时间服务), 239
- clock skew (时钟偏移), 237
- computer clock (计算机时钟), 237
- distributed physical time service (分布式物理时间服务), 242
- Geostationary Operational Environmental Satellite (地面站操作环境卫星), 236
- Network Time Protocol (网络时间协议),

243
 synchronizing (同步), 237
 time provider (时间提供者), 237
 Universal Time Coordinator (宇宙协同时间), 236
 use for total ordering (利用总体排序), 253
 pipe (管道), 61
 mkfifo command (Mkfifo 命令), 64
 mknod command (Mknod 命令), 64
 named pipe (命名管道), 63
 UNIX names pipe (UNIX 名字管道), 64
 Unnamed pipe (非命名管道), 61
 plug and play (即插即用)
 参见 Windows 2000
 polling (轮询), 56, 113, 45, 125
 POSIX, 33, 36
 compliance in Windows 2000 (Windows 2000 中的遵循), 292
 names pipes (名字管道), 64
 PRAM consistency (PRAM 一致性), 218, 219
 precedence relationship (优先关系), 45, 46
 process (进程), 32
 addressing (编址), 59
 distributed process management (分布进程管理), 151
 divisible processes (可分进程), 40
 indivisible processes (不可分进程), 40
 migration (迁移), 41
 orphan process handling (孤儿进程处理), 168
 processor allocation (处理器分配)
 见 scheduling
 process management (进程管理), 38
 scheduling (调度), 44
 process management (进程管理), 38
 processor consistency (处理器一致性), 219

R

race condition (竞争条件), 112

real-time systems (实时系统), 16, 133
 distributed real-time (分布式实时), 17
 embedded real-time (嵌入式实时), 16
 firm real-time (固件实时), 16
 hard real-time (硬件实时), 16
 reactive real-time (交互实时), 16
 soft real-time (软实时), 16
 release consistency (释放一致性), 222, 224
 参见 lazy release consistency (懒惰(延时)一致性)
 acquire access (捕获访问), 222
 barrier (壁垒(障碍)), 222
 protected variable (保护变量), 223
 release access (释放访问), 222
 remote procedure call RPC, 247
 remote procedure call (远程过程调用), 72
 binding (捆绑), 73
 call semantics (调用语义), 75
 data type support (数据类型支持), 72
 in NFS, 197
 in Windows 2000, 298
 in Windows 2000 cluster service (in Windows 2000 集群服务), 316
 marshalling (整理), 73
 parameter type (参数类型), 72
 Sun's ONC RPC, 76
 reparse points (重解析点)
 参见 Windows 2000
 retrieval disparity (检索), 209, 213
 routing (路由), 182
 RPC
 参见 remote procedure call
 RSA (三重公钥加密算法), 266
 in Windows 2000, 290
 three phases of (三相的), 268

S

scheduling (调度), 44
 参见 precedence relationship

- gang scheduling (Gang 调度), 45
 - polled-loop (轮询), 45
 - scheduler organization (调度程序组织), 45
 - thread scheduling (线程调度), 37
 - Secure Socket Layer (安全套接字层)
 - support for in Windows 2000 (在 Windows 2000 中的支持), 319
 - security (安全性), 129, 137, 259
 - access control list (访问控制表), 189
 - asymmetric cryptography (不对称加密), 265
 - capability (能力), 189
 - certificate list (可信度表), 271
 - concerns in distributed system (涉及分布式系统), 270
 - Diffie-Hellman key exchange (Diffie-Hellman 密钥交换), 269
 - digital signature (数字签名), 265
 - distributed access control (分布式存取控制), 259
 - distributed authentication (分布式授权), 259
 - file protection mode (文件保护模式), 188
 - in NTP, 246
 - in Windows 2000, 292
 - in Windows NT, 319
 - Kerberos, 276
 - RSA cryptography RSA 加密, 266
 - Security Reference Monitor in Windows 2000 (Windows 2000 中的安全访问监控), 298
 - semaphore (信号量), 107, 111
 - UNIX semaphore support (UNIX 信号量支持), 110
 - Security Descriptor Definition Language (安全描述定义语言)
 - 参见 Windows 2000
 - Security Support Provider Interface 安全支持提供者界面
 - 参见 Windows 2000
 - semaphore (信号量), 135
 - in Clouds, 132
 - semaphore object in Windows 2000 (Windows 2000 中的信号量对象), 296
 - sequential consistency (顺序一致性), 216, 217, 221, 224
 - shared data (共享数据), 90
 - shared memory (共享内存) 32, 86, 111
 - false sharing (伪共享), 97
 - 参见 distributed shared memory
 - SIMD (单指令流多数据流), 21
 - Simple Network Time Protocol (简单网络时间协议), 247
 - SISD (单指令流单数据流), 21
 - snoopy cache (窃听式高速缓存), 88, 89
 - socket (套接字), 63
 - Java socket (Java 套接字), 69
 - socket (套接字)
 - SSL (安全套接字层), 276
 - UNIX socket (UNIX 套接字), 65
 - Solaris, 76
 - SSL, 276
 - stochastic learning (随机学习), 165
 - strict consistency (严格一致性), 216, 217
 - subject-oriented programming (面向主题程序设计), 130
 - supercomputer (超级计算机), 19
 - synchronization (同步), 235
 - synchronous event (同步事件), 17
 - synchronization variable (同步变量), 220, 224
- T**
- task interaction graph (任务交互图), 40
 - TCP/IP, 8, 11, 67, 69
 - in NTP (网络时间协议), 248
 - IP addressing (IP 编址), 183
 - IP routing (IP 路由), 182
 - IPv4 name resolution (IPv4 名字解析), 184

- security issue (安全状况), 281
 - telecommunication (远程通信), 25
 - thrashing (颠簸), 85
 - distributed (分布式), 97
 - threads (线程), 295
 - client/server paradigm (C/S 方案), 34
 - in Chorus, 135
 - in Clouds, 131
 - Java, 37
 - kernel support (内核支持), 37
 - multithreaded paradigm (多线程范例), 34
 - POSIX, 36
 - specialist paradigm (专用程序范例), 34
 - user-space support (用户空间支持), 38
 - three-point test (三点测试) 106
 - time service (时间服务), 239
 - centralized physical time service (集中式物理时间服务), 239
 - distributed physical time service (分布式物理时间服务), 242
 - logical clock (逻辑时钟), 250
 - Network Time Protocol (网络时间协议), 243
 - timestamp (时间戳), 117, 121
 - counter based (基于计数器), 253
 - in Windows 2000 change journal (Windows 2000 改变日志), 311
 - NTP format (NTP 格式), 246
 - the use of in NTP (NTP 中的使用), 249
 - types in NTP (NTP 中的类型), 249
 - use for global time (使用全局时间), 236
 - use for total ordering (使用整体排序), 253
 - token passing algorithm (令牌传输算法), 118
 - total ordering (整体排序), 254
 - transaction management (事务管理), 175, 207
 - domino effect (多米诺现象), 231
 - lost update (丢失恢复), 208, 213
 - nested transaction (嵌套事务), 228, 229
 - premature read (预读), 230
 - premature write (预写), 230
 - retrieval disparity (恢复不一致), 209, 213
 - rollback (回滚), 228
 - two-phase commit (两阶段提交), 224
 - transaction (事务)
 - 参见 Transaction management
 - transparency (透明性), 15
 - access transparency (访问透明性), 15
 - concurrency transparency (并发透明性), 16
 - failure transparency (故障透明性), 16
 - location (位置), 181, 182
 - location transparency (位置透明性), 15, 131
 - migration transparency (迁移透明性), 15
 - 参见 migration
 - name (名字), 181
 - name transparency (名字透明性), 15, 131, 134
 - parallelism transparency (并行透明性), 16
 - replication transparency (复制透明性), 16, 192
 - two-phase commit protocol (两阶段提交协议), 224
 - commit phase (提交点), 227, 230, 231
 - prepare to commit phase (准备到提交点), 225, 229
 - rollback (回滚), 227
- U**
- UDP (用户数据报协议), 67
 - in NTP, 248
 - UMA architecture (UMA 结构), 19, 85
 - Universal Time Coordinator (调整国际标准时), 236
 - UNIX, 67, 127, 138

directory structure (目录结构), 195
 firewall issue (防火墙概况), 280
 names pipe (名字管道), 64
 pipe (管道, 流水线), 61
 POSIX thread (POSIX 线程), 36
 semaphore support (信号量支持), 110
 shared memory (共享内存), 87
 socket (套接字), 65
 Solaris, 76
 time service (时间服务), 240

V

Very Large Memory (海量内存)
 参见 Windows 2000
 virtual memory (虚拟内存), 82, 86, 98
 in Windows 2000, 298

W

weak consistency (弱一致性), 220, 222, 224
 synchronization variable (同步变量), 220
 Windows 2000, 289
 access control list (访问控制表), 304
 Active Directory (活动目录), 306, 309
 Advanced Configuration and Power Interface (高级配置和电源界面)
 参见 windows 2000
 Advanced Power Management (高级电源管理), 300
 Advanced Server Edition (高级服务器版本), 290
 C2 certification (C2 认证), 293
 change journal (改变日志), 309
 Cluster Service (集群服务), 316
 cluster service architecture (集群服务体系结构), 317
 Datacenter Server Edition (数据中心服务器(编辑)版本), 290
 design goal (设计目标), 290

device driver (设备驱动程序), 297
 Encrypting File System (加密文件系统)
 参见 Windows 2000
 executive (执行), 293, 297
 HAL (硬件抽象), 289
 Hardware Abstraction Layer (硬件抽象层), 297
 参见 Windows 2000
 indexing process (索引处理), 314
 Kernel (内核), 293, 299
 kernel mode (内核模式), 291
 kernel object (内核对象), 295
 LDAP (轻量级目录存放协议), 308
 Microsoft Index Server (MS 索引服务器), 311
 Microsoft Management Console (MS 管理控制台), 290, 313
 migration within a cluster (集群中的迁移), 317
 MMC extension type (MMC 扩充类型), 315
 MMC snap-ins (MMC 插件), 315, 321
 name space (名词空间), 308
 NTFS (NT 文件系统), 303
 Object Manager (对象管理器), 298
 plug and play (即插即用), 289, 290
 Professional Edition (专业版本), 290
 remote storage handing (远程存储处理), 306
 reparse point (重解析点), 304, 305
 Security Configuration Editor (安全配置编辑器), 319
 Security Support Provider Interface (安全支持提供者界面), 319
 Server Edition (服务器版本), 290
 subsystem (子系统), 292
 update sequence number (更新序列号), 310
 use of X.500 (X.500 的应用), 306
 Very large Memory (海量内存)

- 见 Windows 2000
- Win 32, 299
- Windows NT, 36
- 见 Windows 2000
- Windows NT
- 参见 Windows 2000
 - security (安全性), 319
- X
- X.500, 200
 - Directory Model (目录模型), 202
 - in Windows 2000, 306
 - relationship to Active Directory (与活动目录的关系), 308
- X.509, 272
 - multiple certificate authorities (多重确认授权), 274
 - one-way authentication (one-way 授权 (单路)), 273
 - three-way authentication (three-way 授权 (三路)), 273
 - two-way authentication (two-way 授权 (二路)), 273